Java CAPS 6 Update 1
Invoking MTOM Web Service
using Java CAPS Classic Web Service Client

Michael.Czapski@sun.com
February 2009

Contents

1. Introduction

If we overlook the fact that using web services to transfer large payloads is a very stupid idea, we will be faced with the need to implement the optimisation mechanisms to make transfer of large payloads using web services a little less inefficient from the stand point of the size of the over-the-wire data to be transferred. The standardised, supported mechanism for this is the Message Transmission Optimisation Method (MTOM), http://en.wikipedia.org/wiki/MTOM. Java CAPS Repository-based Web Services don't offer a convenient mechanism to provide MTOM support.

This note walks through the implementation of a Java CAPS Repository-based, eInsight-based web service consumer and the implementation of the EJB-based Web Service Wrapper Consumer for this service, which provides support for MTOM. The Note discusses how to exercise the wrapper service using the NetBeans web services testing facilities, how to trigger the Java CAPS Repository-based web service invoker and how to observe on-the-wire message exchanges. The invoker implementations discussed in this Note will invoke the web service providers discussed in an earlier Note, "Java CAPS - Exposing MTOM-capable Java CAPS Classic Web Service", http://blogs.sun.com/javacapsfieldtech/entry/java_caps_exposing_mtom_capable.

2. Create a Project Group

I find the NetBeans flat project structure annoying. The Enterprise Designer hierarchical project structure, ported to Java CAPS 6 Classic, was a much more reasonable way of organizing projects and project artefacts. The Project group feature of NetBeans is a poor substitute for that. Be it how it may, I have gotten into a habit of creating project groups so I can collect related projects and open/close related projects in a hit.

Reuse the project group created when developing the projects discussed in an earlier Note, “Java CAPS - Exposing MTOM-capable Java CAPS Classic Web Service”, http://blogs.sun.com/javacapsfieldtech/entry/java_caps_exposing_mtom_capable.

3. WSDL and Services Notes

Java CAPS 6 is fussy about the kind of WSDL it will accept for different kinds of projects. In this Note we will be dealing with EJB-based Web Services and Repository-based Web Services. Experience tells me that the only kind of WSDL that is acceptable to both is a) document/literal WSDL and b) a WSDL with an in-line XML Schema. Another words, don't use an external XML Schema included in the WSDL and expect it to work. It will work in same projects but not in others. In particular, if you create a Classic Web Service, which you wish to invoke form an EJB via a Web Service Reference, the tooling will be unable to find the referenced XML Schema and the result will be a fiasco.

As we work through this Note we will invoke the service exposed by the solution discussed in the previous Note in this set of Notes, "CAPS - Exposing MTOM-capable Java CAPS Classic Web Service", already referenced in the Introduction. We will not create a new WSDL but rather will use an existing WSDL, provided by the Web Service which we need to invoke.

4. Build eInsight / BPEL 1.0-based Web Service Client

We will develop and exercise the client solution in stages.

In Stage 1 we will create a Repository-based Web Service Client that will communicate with the Repository-based Web Service Provider, developed in the previous Note.

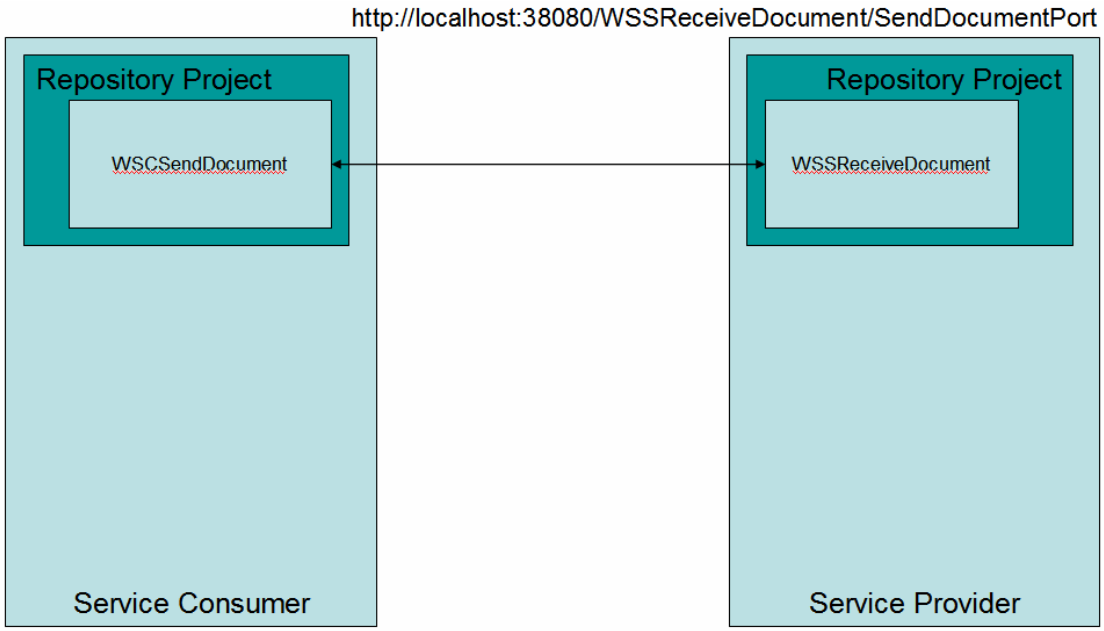

Figure 4-1 Stage 1 – Repository-based services

Let's create a Java CAPS Repository-based / Classic project, WSCSendDocument. Figures 4-2 and 4-3 illustrate the steps.

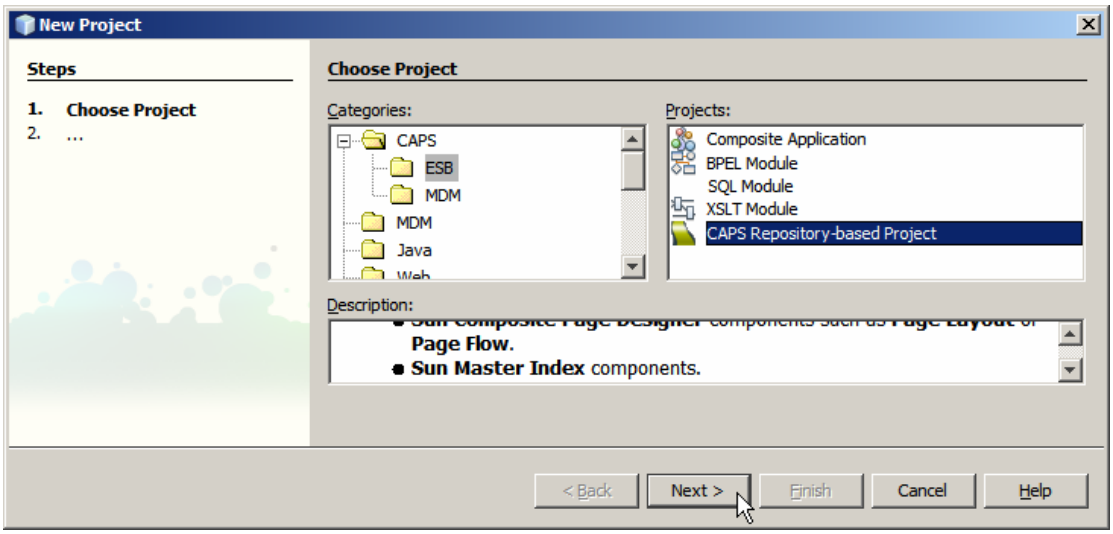

Figure 4-2 Creating Java CAPS Repository-based Project

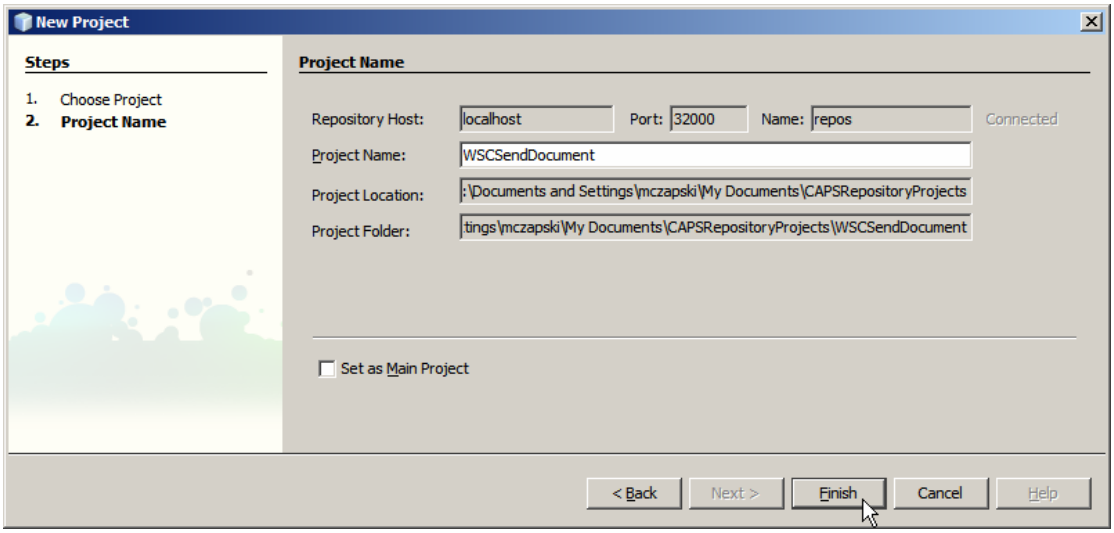

Figure 4-3 Naming the project

We will pretend that the service we need to invoke is a 3rd party web service and the only way to access the WSDL is to ask for it via a URL.

Let's now import the Web Service Definition (WSDL) from the service we are to invoke. Before starting the Wizard let's copy to clipboard the URL pointing to the WSDL document. Start the UDDI Browser Web Application, typically at http://{JC6UDDIHost}:{JC6UDDIPort}/CAPSUDDI/uddibrowse.jsp, and copy the location of the WSDL to the clipboard.

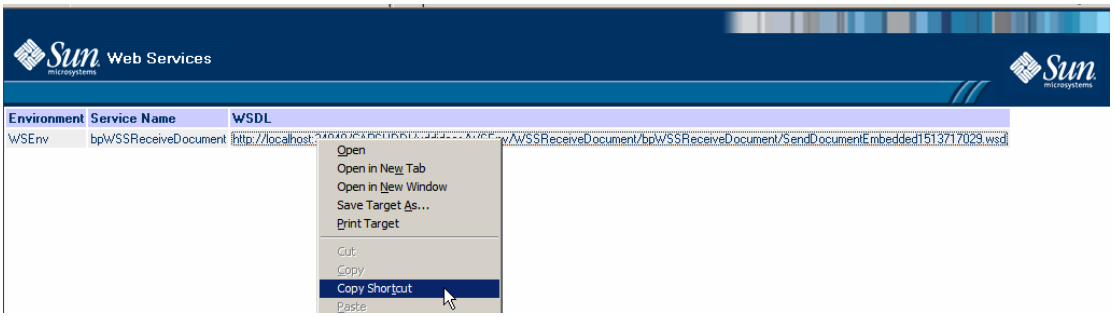

Figure 4-4 Copy WSDL URL to the clipboard

Right-click the name of the Classic project, WSCSendDocument, and choose Import->Web Service Definition. Figure 4-5 illustrates this.

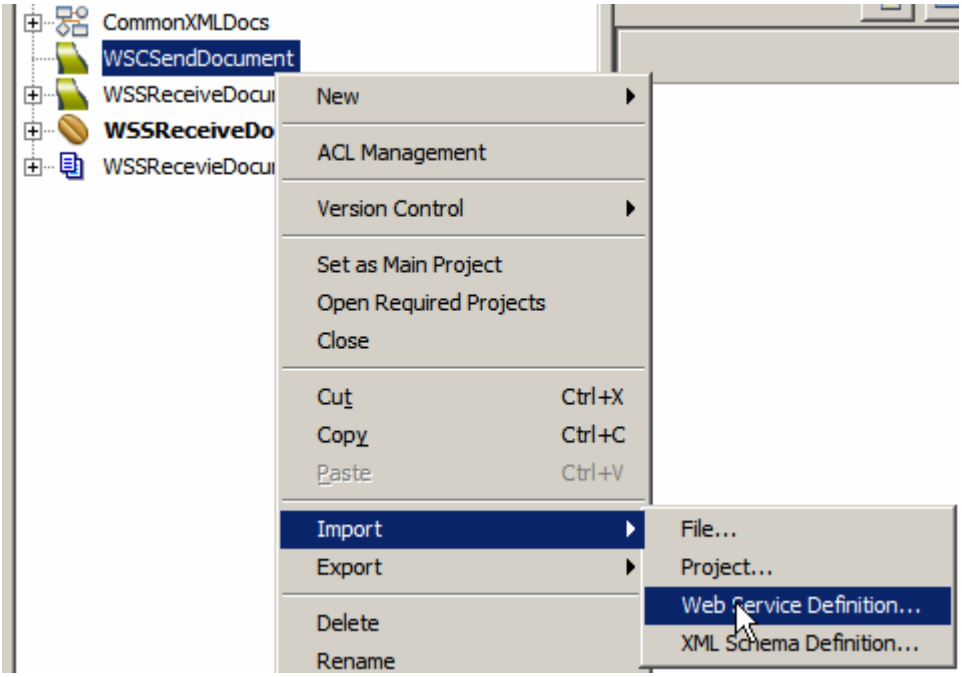

Figure 4-5 Import WSDL Wizard – step 1

Change the location to URL, paste the URL into the text box, click Add and click Next. This is illustrated in Figure 4-6.

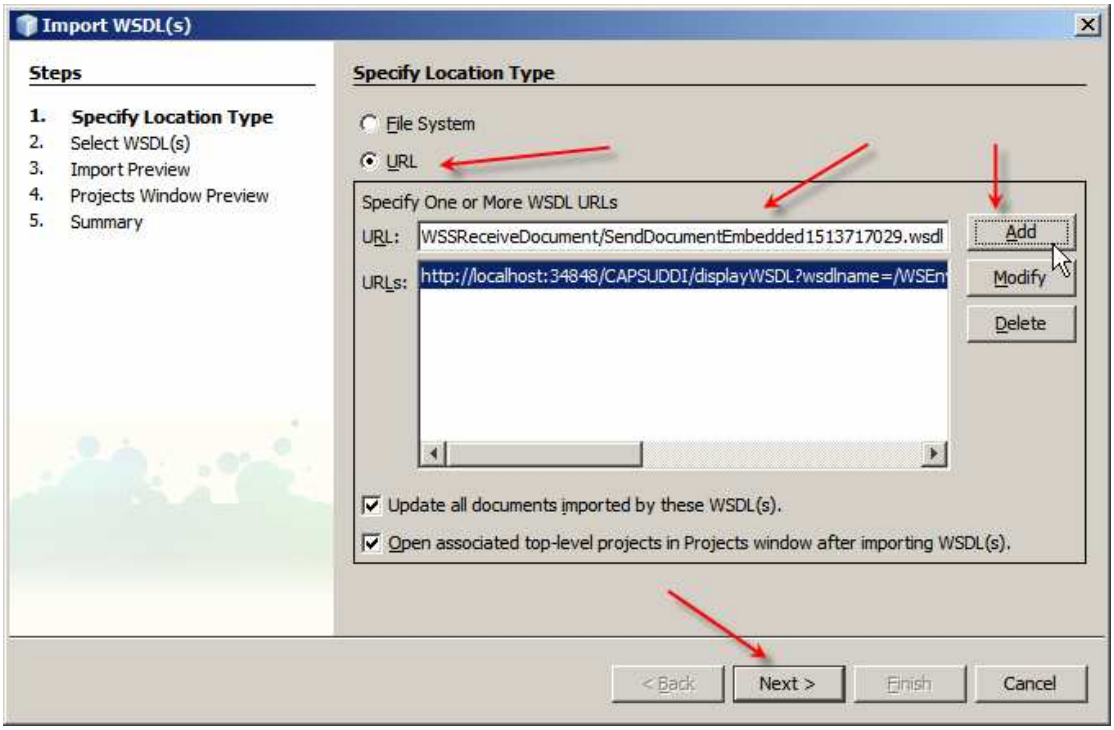

Figure 4-6 Specify location of the WSDL

Accept what is offered as illustrated in Figure 4-7.

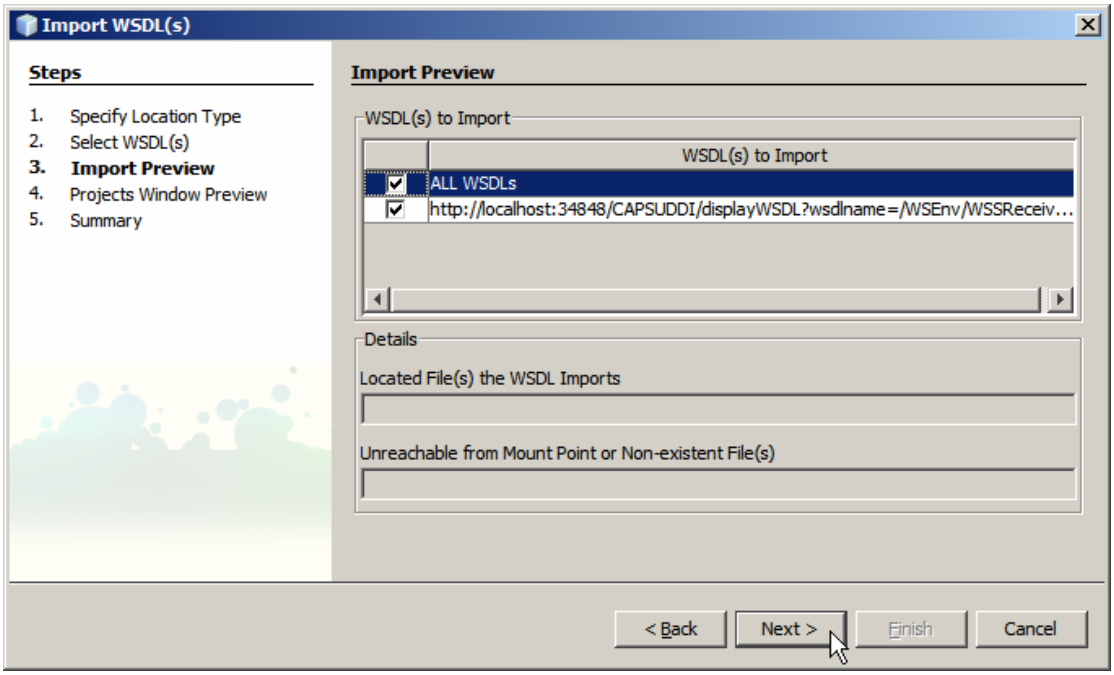

Figure 4-7 Accept what is offered

Accept what is offered by clicking Next, as illustrated in Figure 4-8.

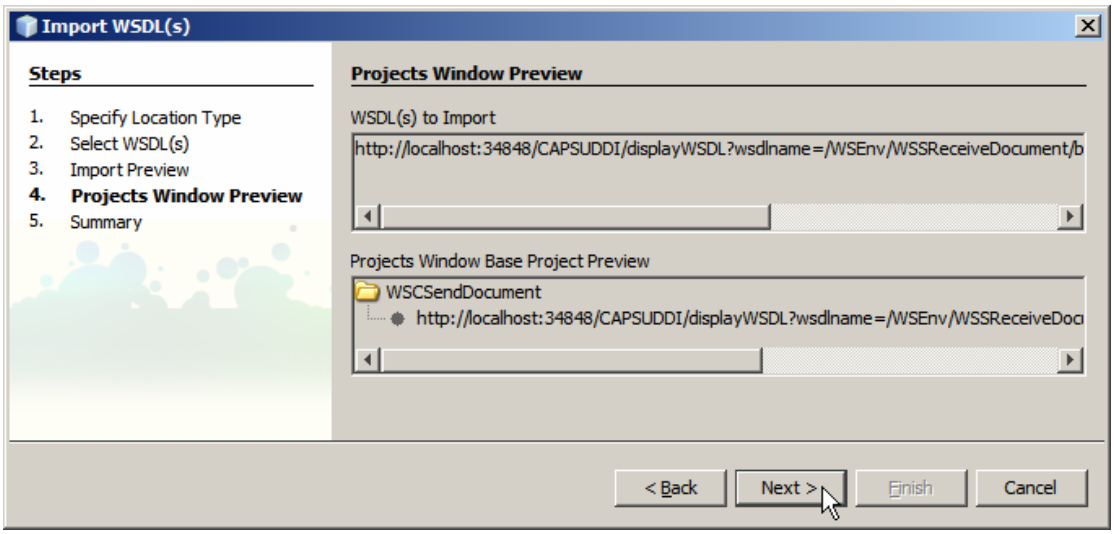

Figure 4-8 Accept what is offered

Assuming no errors or warnings are present, click Finish, as illustrated in Figure 4-9.

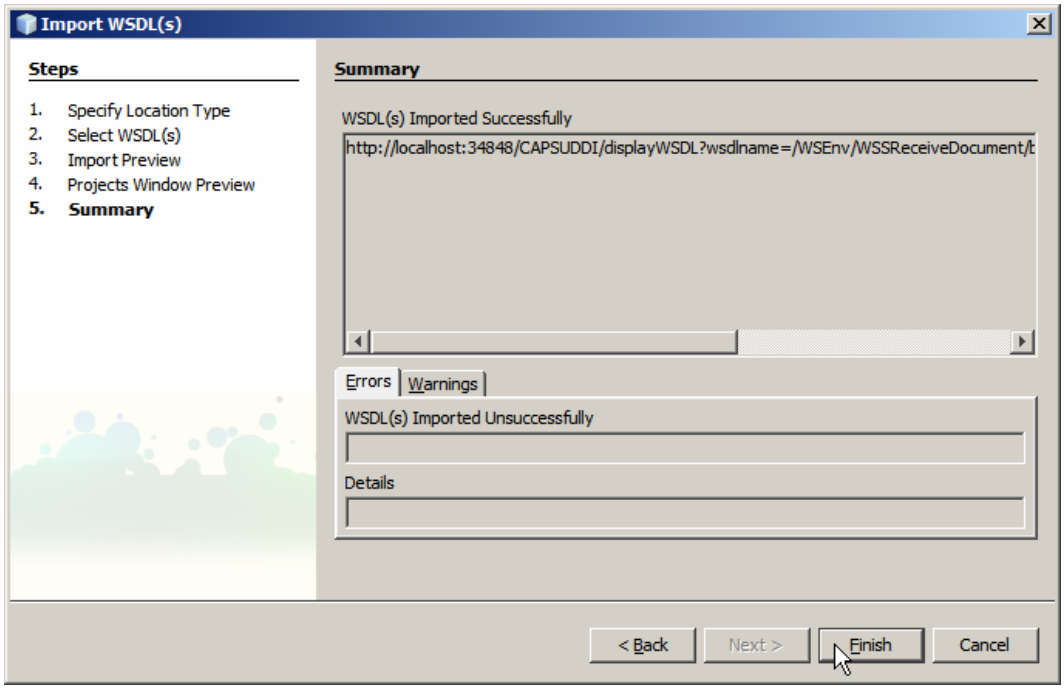

Figure 4-9 Complete Wizard

The process above produces a Classic WSDL artefact, shown in Figure 4-10.

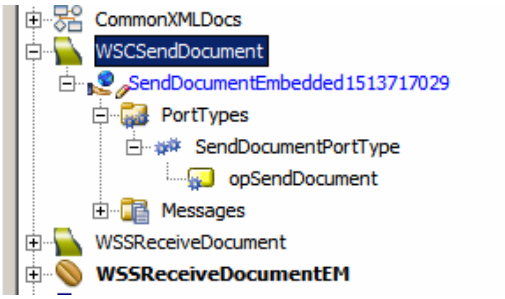

Figure 4-10 Classic WSDL artefact

Create a new Business Process, bpWSCSendDocument. Figure 4-11 illustrates the menu options involved.

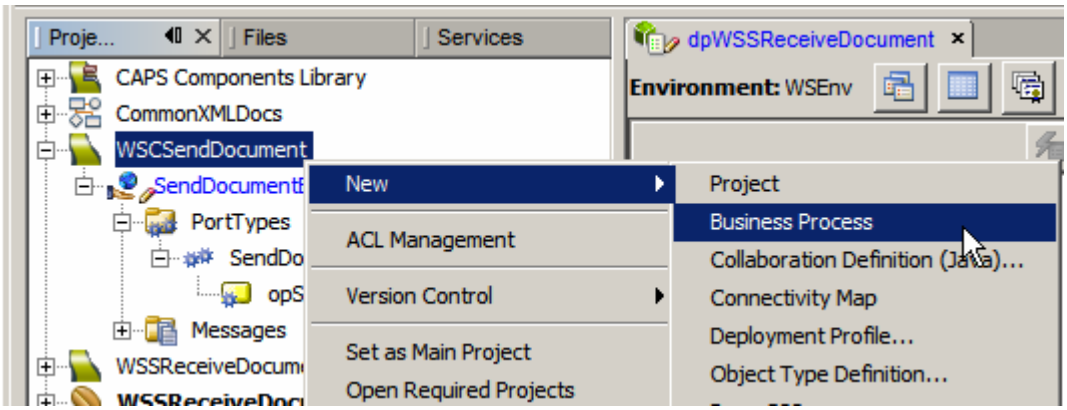

Figure 4-11 Create a new Classic Business Process

The web service client will be triggered by the arrival of a JMS message and will send the web service invocation response to a JMS destination.

Let’s expand the CAPS Components Library -> Enterprise Service Bus -> JMS node tree and drag the receive and the send services to the business process canvas, as shown in Figure 4-12.

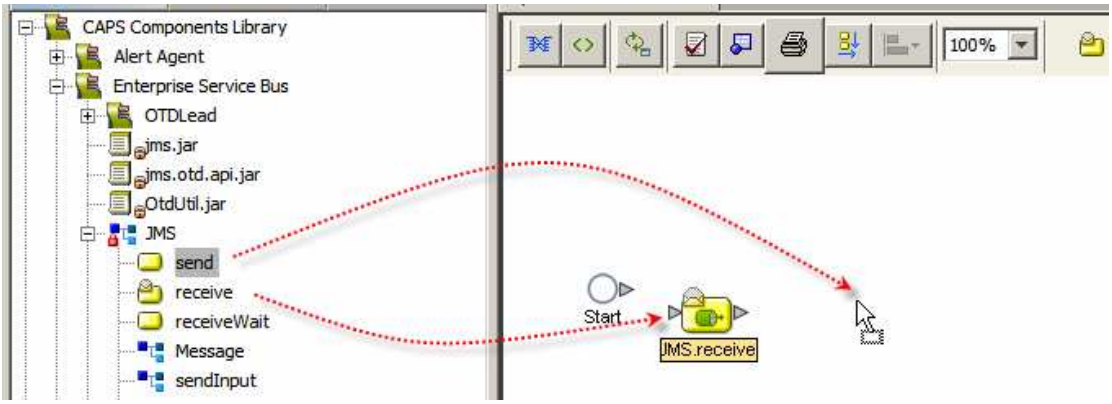

Figure 4-12 Add JMS receive and JMS send activities to the process

To implement the service let’s drag the Web Service operation onto the business process canvas and choose the “Invoke …” option. Figures 4-13 and 4-14 illustrate the steps.

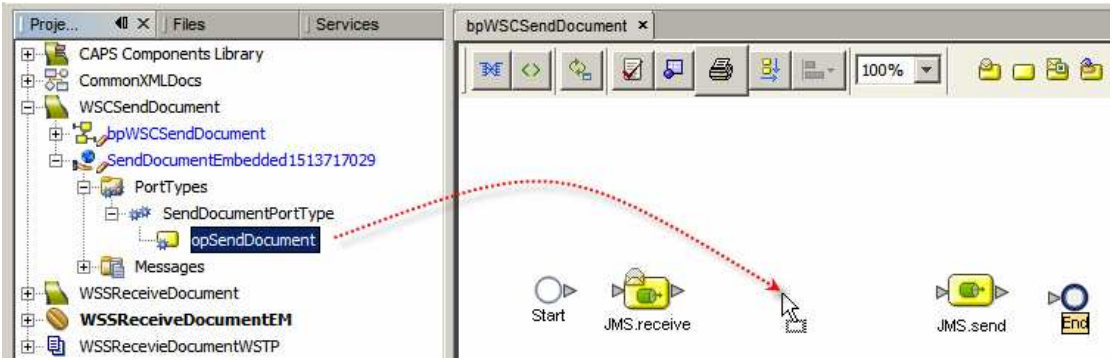

Figure 4-13 Drag the web service operation onto the BP canvas

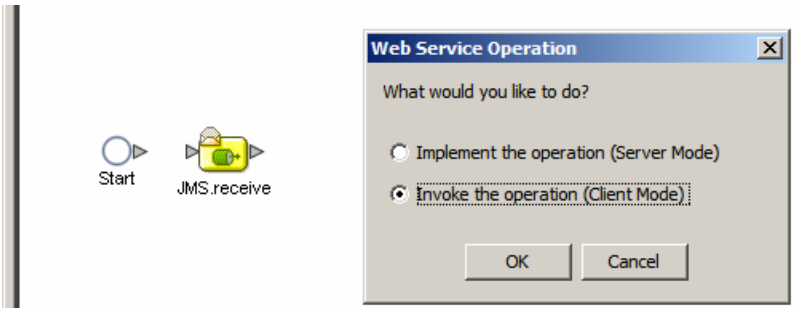

Figure 4-14 Choose “Invoke …”

Connect the activities together. Add a Business Rule between the JMS receive and the Web Service invocation activity and map as shown in Figure 4-15. To keep the project simple we will map the JMS TextMessage to the DocID node and provide literals or derived literals for the other nodes.

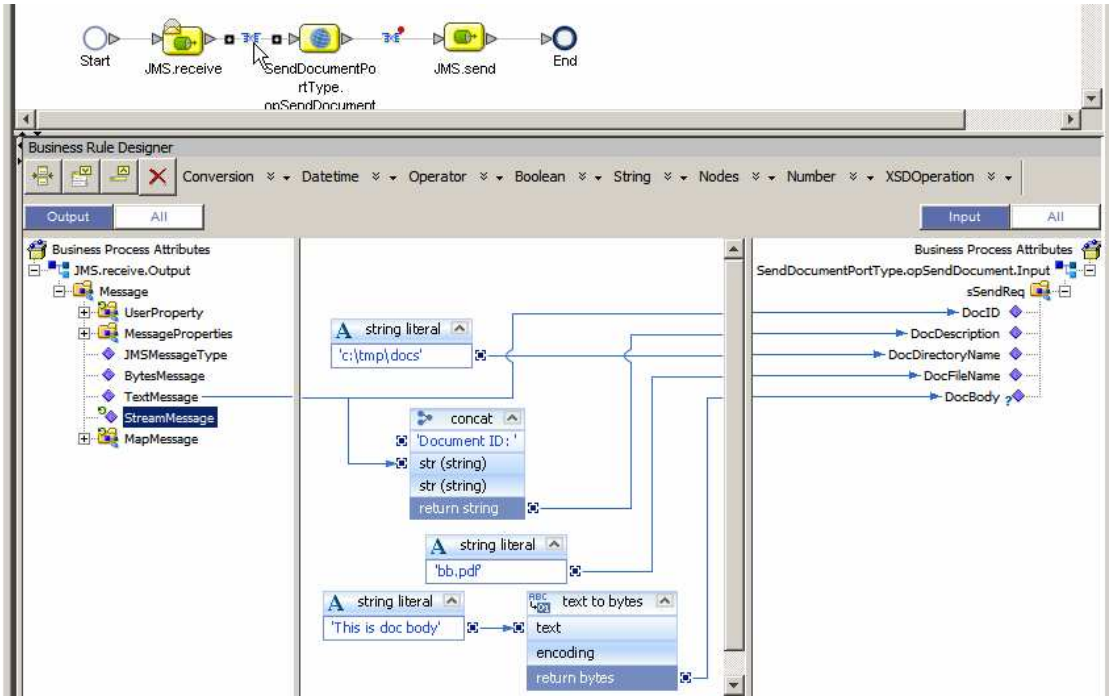

Figure 4-15 Map data to Web Service request

The implementation is deliberately simple since the business logic does not really matter in this case.

Let's now map the web service response to the JMS TextMessage, as illustrated in Figure 4-16.

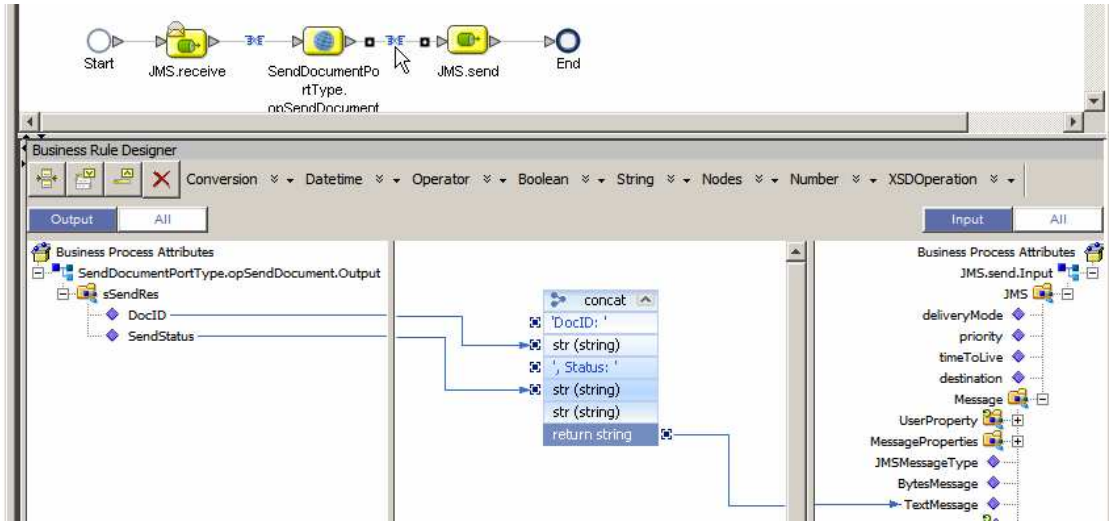

Figure 4-16 Map web service response to the JMS TextMessage

Let's create the Connectivity Map, cmWSCSendDocument, drag the business process onto the CM canvas, add the Web Service Connector, add two JMS Queues, qWSCSendDocumentIn and qWSCSendDocumentOut, and connect. Figure 4-17 shows the completed connectivity map.

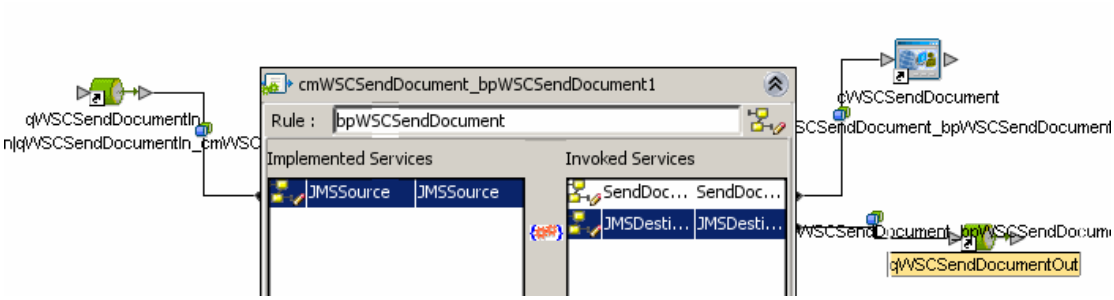

Figure 4-17 cmWSCSendDocument Connectivity Map

Assume you have created a Java CAPS Environment as discussed in Section 13, “

Create Java CAPS Environment".

Add a new SOAP/HTTP Web Service **Client** External System container, WSCSendDocument, and configure it as appropriate to your environment. Figures 4-18 through 4-20 illustrate this for my environment.

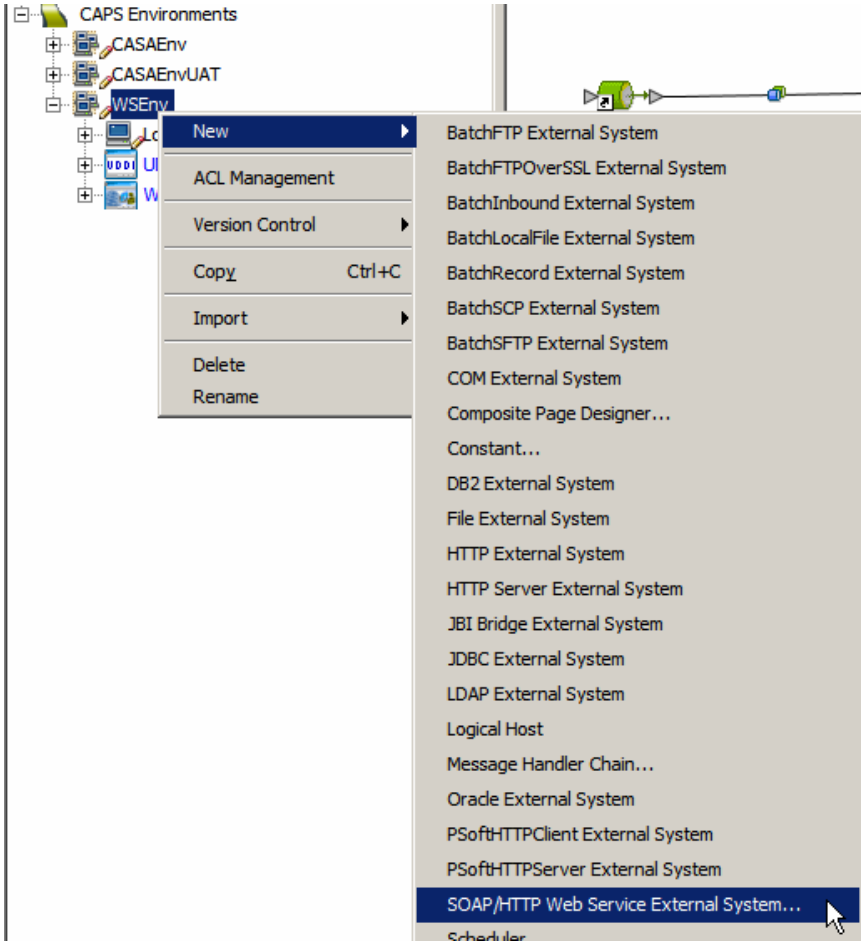

Figure 4-18 Create a SOAP/HTTP Web Service External System …

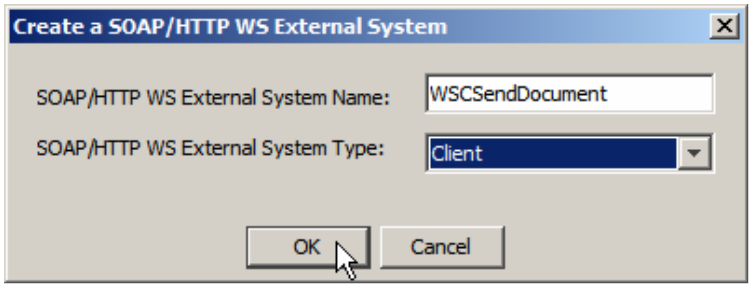

Figure 4-19 Name the container and choose container type

Leave the container properties not configured, as shown in Figure 4-20. The properties will be configured by the build process based on the values provided in the WSDL.

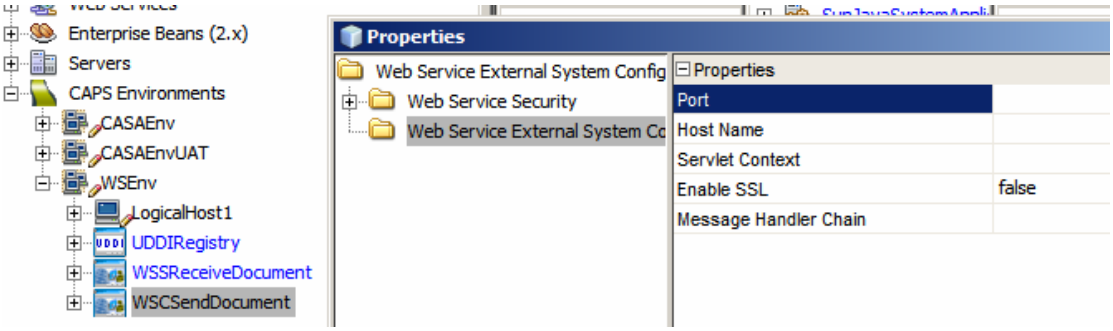

Figure 4-20 Leave the container properties unconfigured

The Java CAPS Environment now has all the containers necessary to deploy the project we have been working on so far. Let's create the Deployment Profile, dpWSCSendDocument, build and deploy the project. Figure 4-21 illustrates a step in this process.

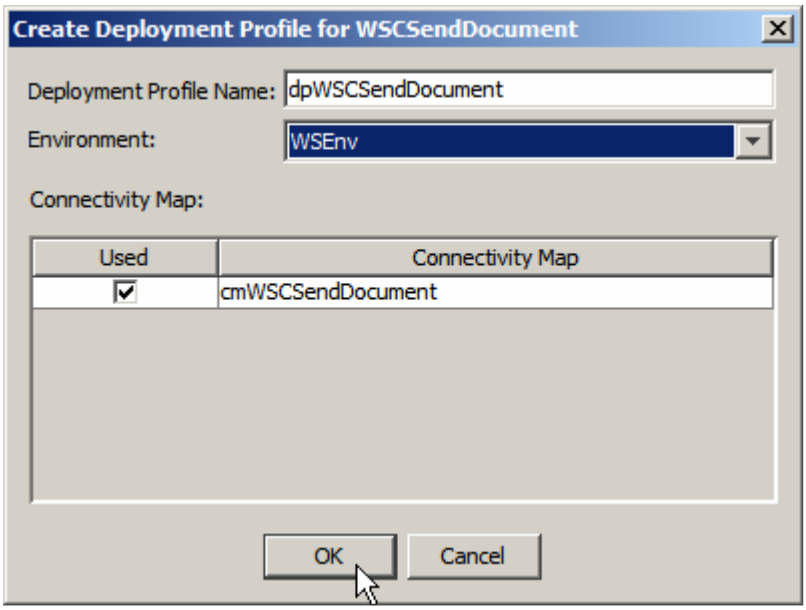

Figure 4-21 Choose Deployment Profile name and the environment to which it will belong

The service is deployed. Now let's exercise it to see if it works.

5. Exercise the Repository Web Service Client - 1

The Business Process we developed in the previous section is triggered by the arrival of a JMS message.

The Web Service client invokes the Repository-based Web Service provider implemented in the previous Note. Neither supports MTOM.

Let's start up the Java CAPS 6 Enterprise Manager web interface, typically at http://localhost:15000, though the port will be set at Java CAPS installation time. For me it is actually http://localhost:32100.

Let's log in, expand the project tree all the way to the connectivity map cmWSCSendDocument and click on the connectivity map name. Figure 5-1 illustrates this.

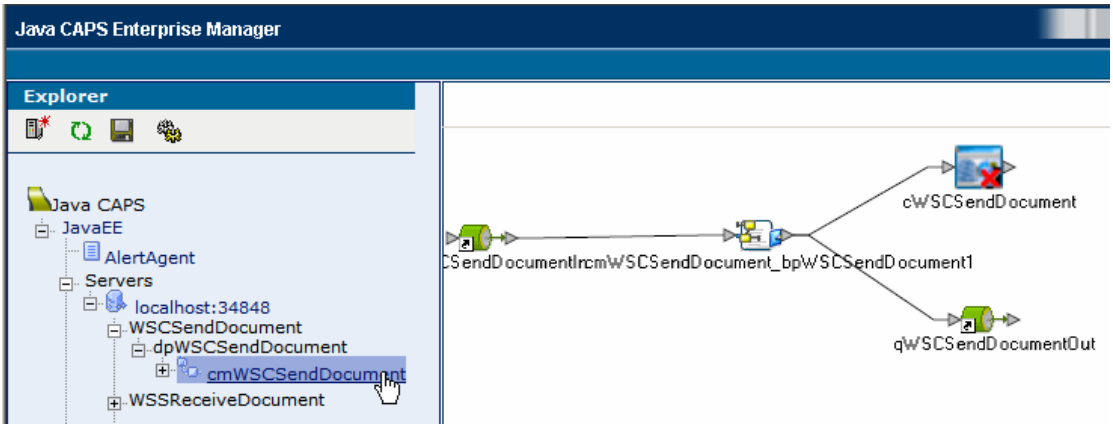

Figure 5-1 Locate connectivity map for project WSCSendDocument

Click on the qWSCSendDocument and click on Send/Publish toolbar button as shown in Figure 5-2.

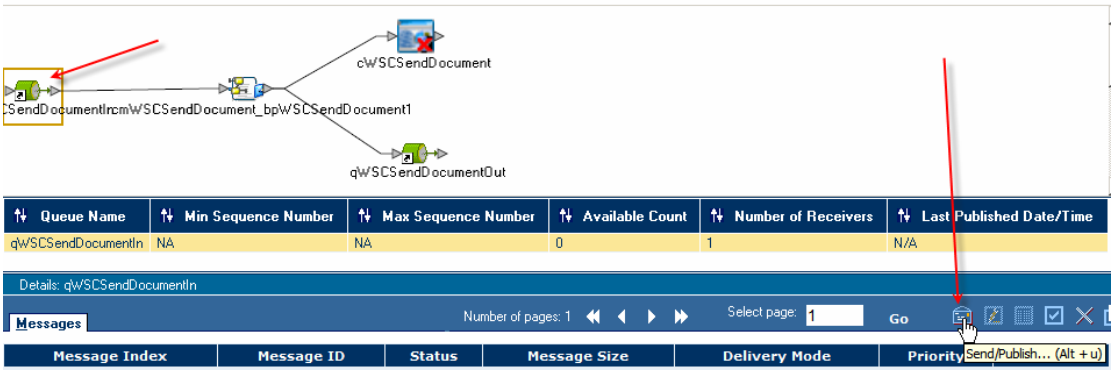

Figure 5-2 Trigger the Send/Publish dialogue

Enter a document ID, say 2, and click the Submit button. Figure 5-3 illustrates this.

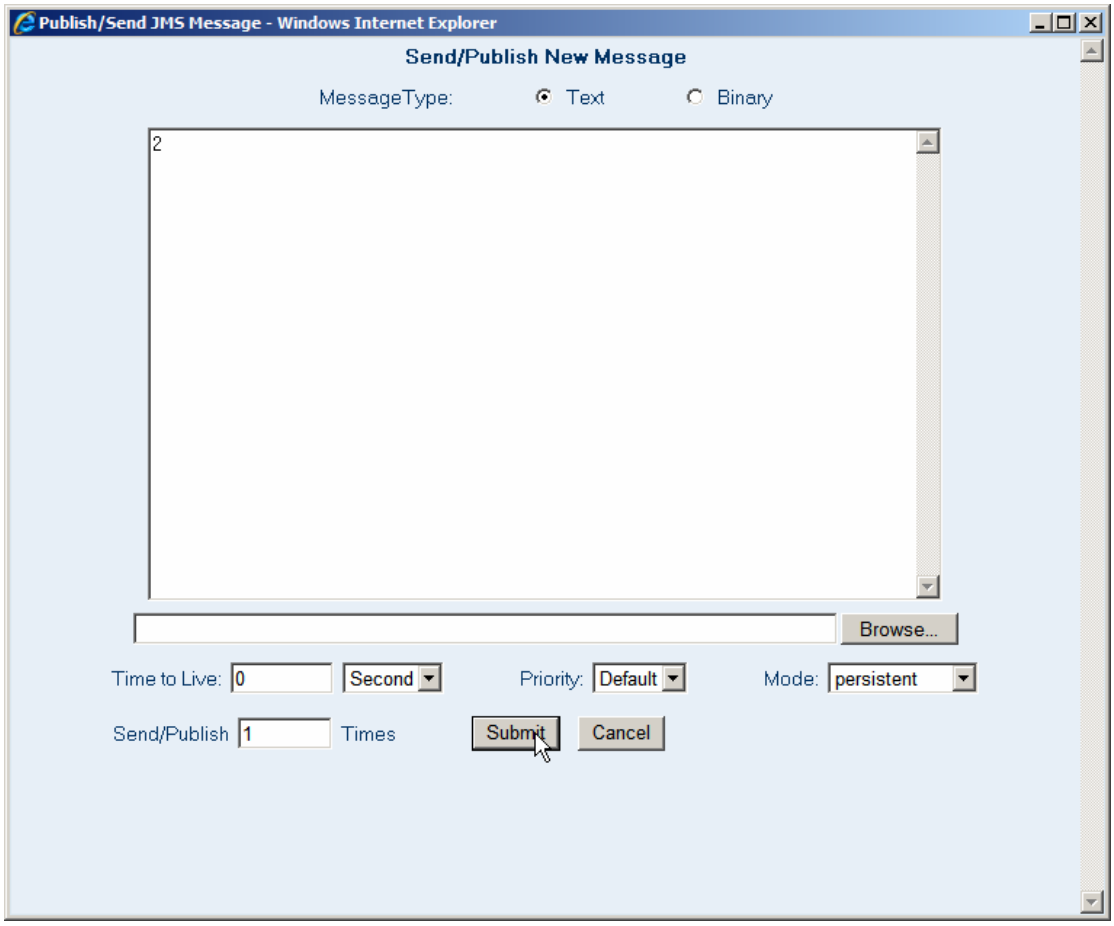

Figure 5-3 Submit a JMS message

Once the message is processed a response will be available in the JMS Queue qWSCSendDocuemtnOut. Select this queue, select the message and click the View/Edit toolbar button as shown in Figure 5-4.

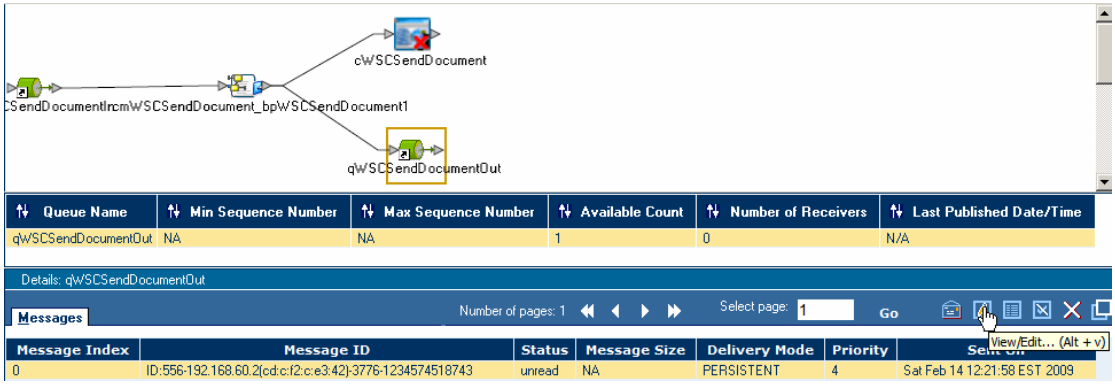

Figure 5-4 Trigger the View/Edit message functionality

Note that the message indicates success, see Figure 5-5.

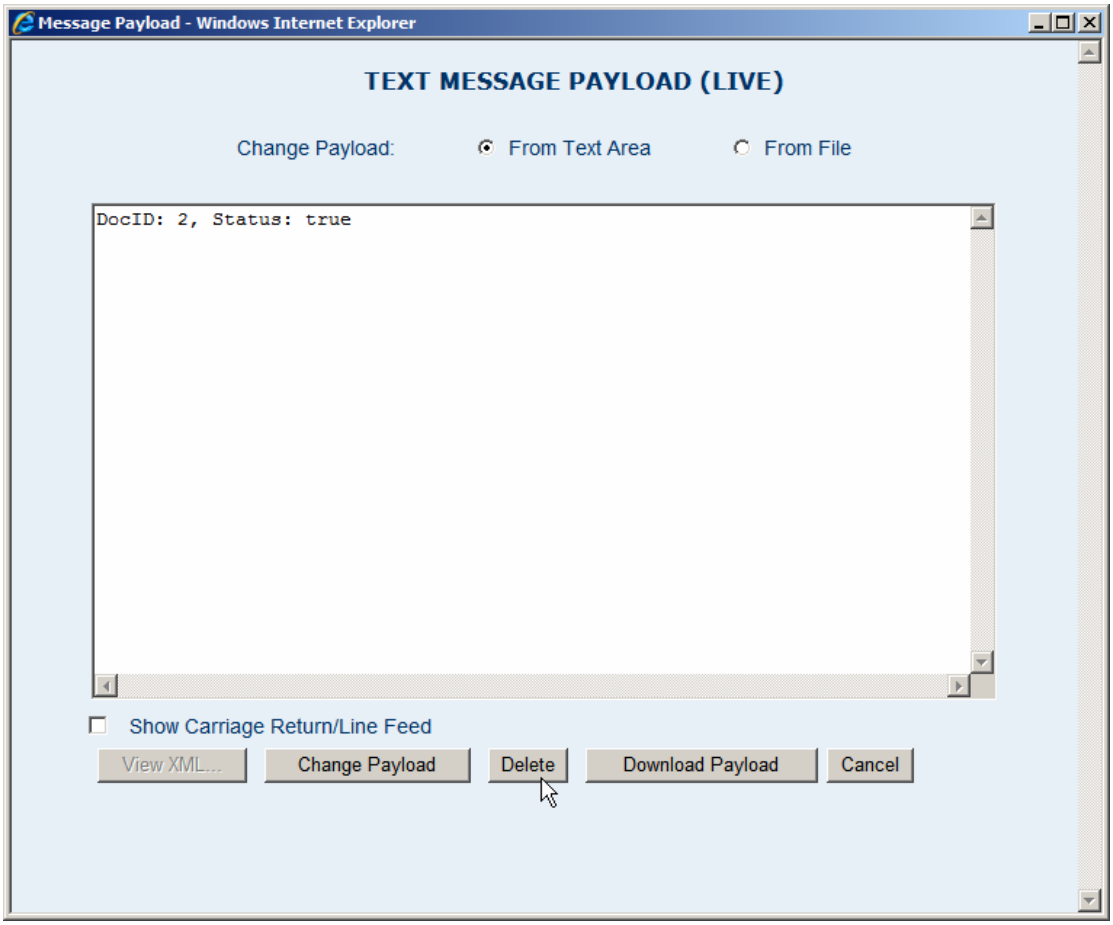

Figure 5-5 Success message

To verify that MTOM functionality was not used let's have a look at what the request and the response look like on-the-wire.

Let's now start the TCP Mon as a proxy listening on port 8888, see section 14, "

Obtain and use the Apache TCP Mon", and configure Web Service Client External System Container to use the proxy, as shown in Figure 5-6.

Figure 5-6 Configure Proxy properties for the WSCSendDocument container

Note that for some obscure reason User Name and User Password properties must be valued even though the proxy does not actually require authentication.

Let's build and deploy the project again, make sure the TCP Mon is running and submit a message as we did before.

The response message shows success. Let's switch to the TCP Mon window and look at the request and the response. Figure 5-7 illustrates the request and the response I have, reformatted slightly to enhance readability.

```
POST http://localhost:38080/WSSReceiveDocument/SendDocumentPort HTTP/1.1
Content-Type: text/xml; charset=utf-8
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Content-Length: 625
SOAPAction: ""
Proxy-Authorization: Basic YTph
User-Agent: Java/1.6.0_10
Host: localhost:38080
Proxy-Connection: keep-alive

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns0="uri:Sun:Michael:Czapski:XSD:SendDocument" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<env:Body>
<ns0:DocumentReq xmlns:tns="uri:Sun:Michael:Czapski:XSD:SendDocument">
<tns:DocID>1</tns:DocID>
<tns:DocDescription>Document ID: 1</tns:DocDescription>
<tns:DocDirectoryName>c:\tmp\docs</tns:DocDirectoryName>
<tns:DocFileName>bb.pdf</tns:DocFileName>
<tns:DocBody>VGhpcyBpcyBkb2MgYm9keQ==</tns:DocBody>
</ns0:DocumentReq>
</env:Body>
</env:Envelope>
```

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
SOAPAction: ""
Content-Type: text/xml;charset=utf-8
Transfer-Encoding: chunked
Date: Sat, 14 Feb 2009 01:34:23 GMT

1cb
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" :
0
```

Figure 5-7 DocBody contains Base64-encoded data – no MTOM optimisation was applied

No MTOM optimisation was applied. The DocBody node contains Base64-encoded document body.

6. Exercise Repository-based Client - 1

In Stage 2 we will point our Repository-based Web Service client at the MTOM Wrapper service, developed in the previous Note. The schematic in Figure 6-1 illustrates this.

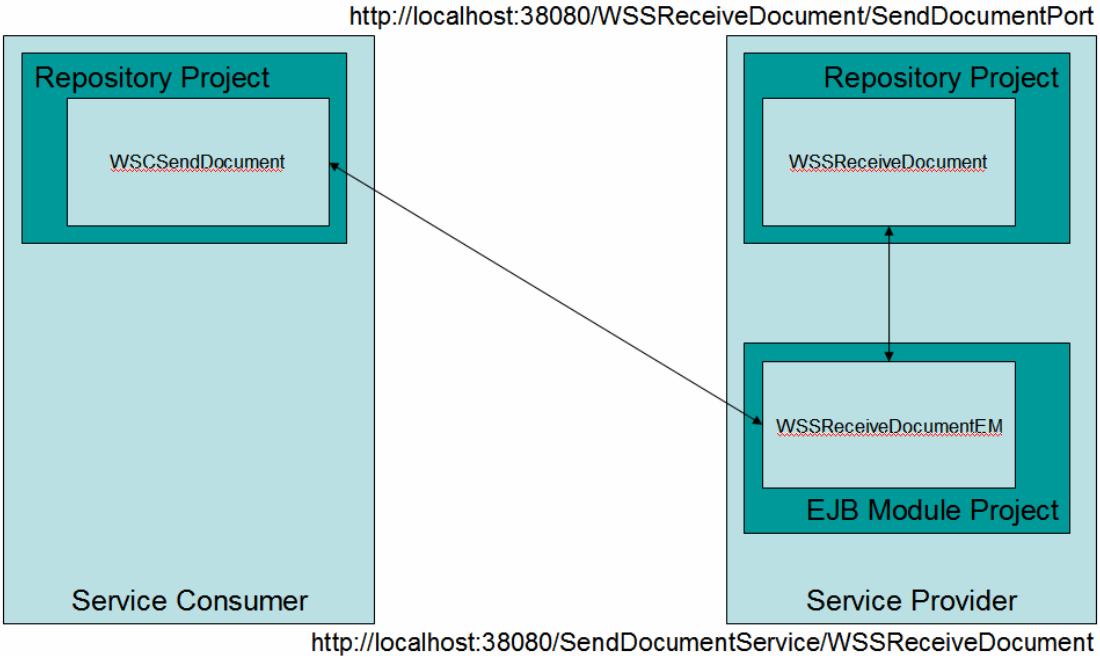

Figure 6-1 Invoking MTOM-capable service

Let's obtain the URL of the MTOM-capable service proxy we developed in the previous Note. Normally the URL would be provided and would point at the service. We have the service implementation, created in the previous Note. Let's open the GlassFish Application Server Admin Console, typically at http://localhost:4848, for me at http://localhost:34848, expand the Web Services node to see the list of Web Services deployed to the Application Server. Figure 6-2 illustrates this.

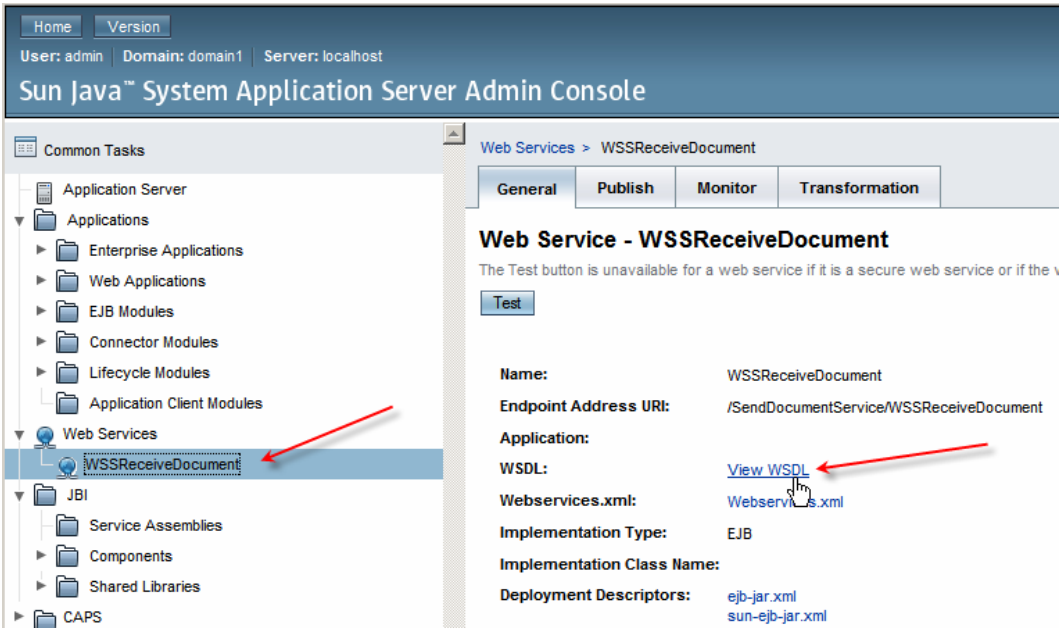

Figure 6-2 Locating WSDL URL

Of the WSDL URL let's copy just the servlet context to the clipboard. Figure 6-3 illustrates this.

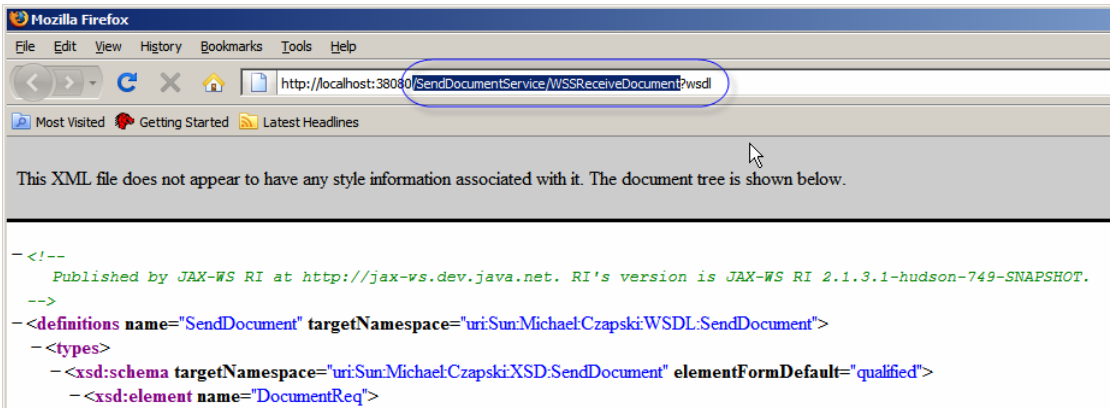

Figure 6-3 Copy Servlet Context to the clipboard

Let's switch to NetBeans, expand the CAPS Environment, open the properties of the WSCSendDocument External System Container and paste the value from the clipboard to the Servlet Context text field, as shown in Figure 6-4.

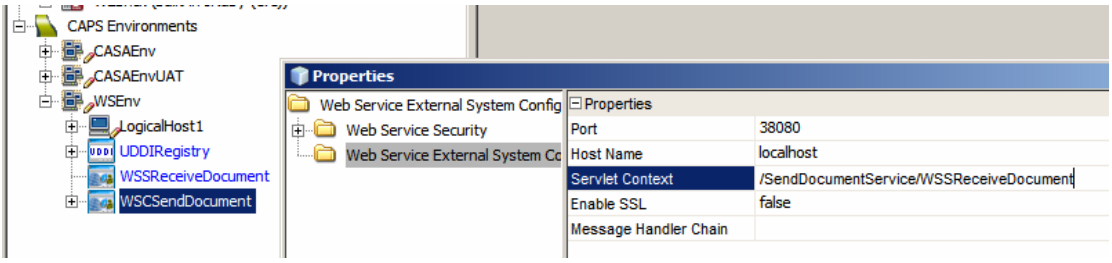

Figure 6-4 Updating servlet context with /SendDocumentService/WSSReceiveDocument

Let's now build, deploy and exercise the project.

The solution still works, in that we received a successful response. Since we left the TCP Mon running, and left the HTTP Proxy settings configured from the previous test we have the request and the response, as seen on the wire, in the TCP Mon window. Slightly re-formatted for readability, they look like these shown in Figure 6-5.

```
POST http://localhost:38080/SendDocumentService/WSSReceiveDocument HTTP/1.1
Content-Type: text/xml; charset=utf-8
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Content-Length: 625
SOAPAction: ""
Proxy-Authorization: Basic YTph
User-Agent: Java/1.6.0_10
Host: localhost:38080
Proxy-Connection: keep-alive

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="uri:Sun:Michael:Czapski:XSD:SendDocument"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<env:Body>
<ns0:DocumentReq xmlns:tns="uri:Sun:Michael:Czapski:XSD:SendDocument">
<tns:DocID>4</tns:DocID>
<tns:DocDescription>Document ID: 4</tns:DocDescription>
<tns:DocDirectoryName>c:\tmp\docs</tns:DocDirectoryName>
<tns:DocFileName>bb.pdf</tns:DocFileName>
<tns:DocBody>VGhpcyBpcyBkb2MgYm9keQ==</tns:DocBody>
</ns0:DocumentReq>
</env:Body></env:Envelope>
```

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Sat, 14 Feb 2009 03:52:09 GMT

ed
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<DocumentRes xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument">
<DocID>4</DocID>
<SendStatus>true</SendStatus>
</DocumentRes>
</S:Body>
</S:Envelope>
0
```

Figure 6-5 Request and Response on-the-wire

There is no doubt that the correct service was invoked because the POST request says so.

7. Build EJB-based Web Service Client Wrapper

To provide MTOM support at the client side we need to build an EJB-based web service client, so we can enable MTOM support. We have an existing Java CAPS Repository-based Web Service. We need to provide an EJB Web Service implementation that that Repository-based service will invoke instead of the web service it was originally configured to invoke. This proxy service will, in due course, provide MTOM support and will invoke the real web service on behalf of the Repository-based web service. So, we need a service implementation and a service client.

This is Stage 3 of the development. The schematic is shown in Figure 7-1.

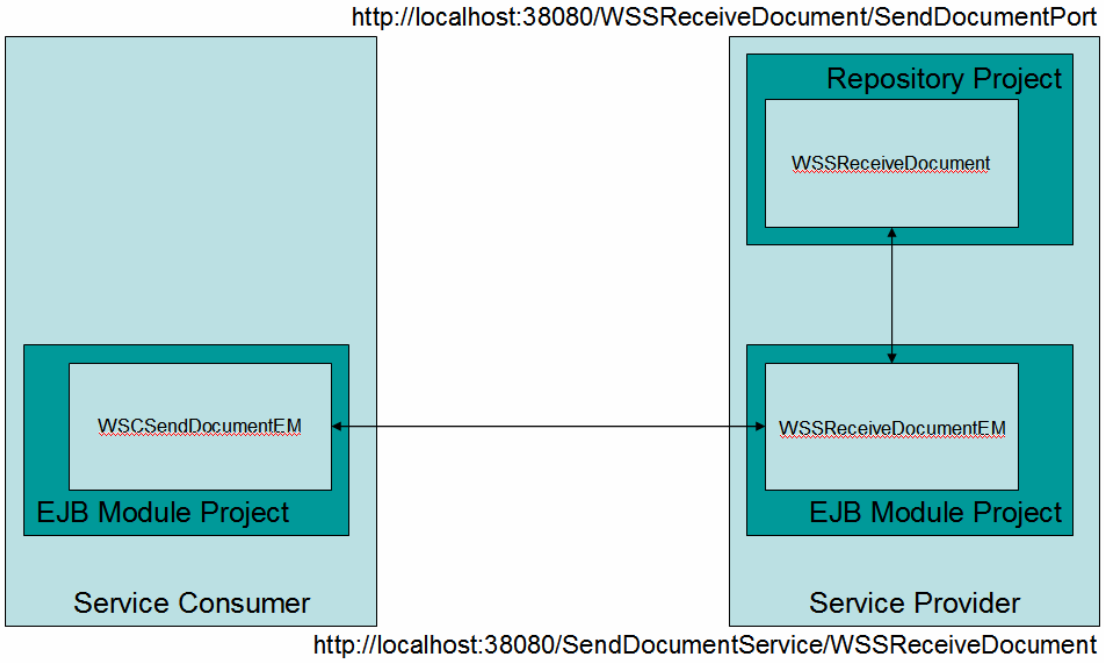

Figure 7-1 Stage 3 – MTOM-enabled Web Service Client Proxy

Let's create an EJB Module project, WSCSendDcumentEM.

Let's open the GlassFish Application Server Console, typically at http://localhost:4848, for me http://localhost:34848, navigate to the Web Services node, click on the WSSReceiveDocument web service name and copy WSDL URL to clipboard. Figure 6.2 illustrates this. We need this WSDL to create a local service proxy. We will modify the WSDL so there is no clash between the service created using it and the service the proxy will invoke, which presented the original WSDL.

In the WSCSendDcumentEM project create a new XML -> External WSDL Document, as shown in Figure 7-2.

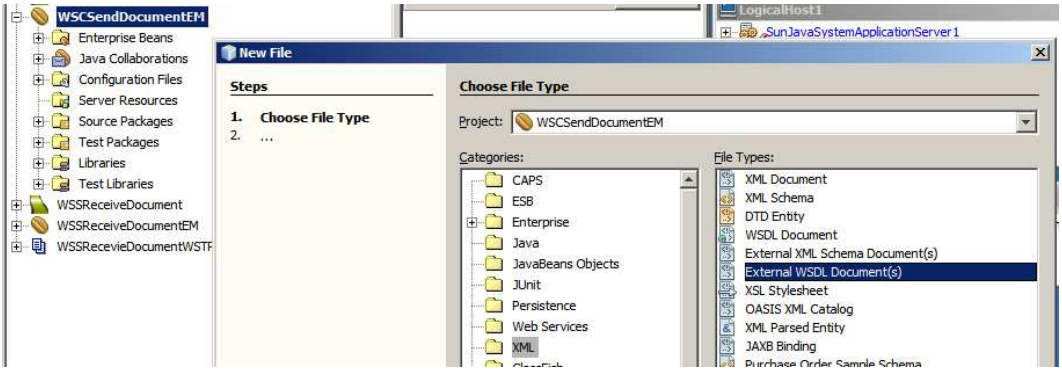

Figure 7-2 Create new External WSDL Document

Provide the URL of the remote service WSDL. Figure 7-3 illustrates this.

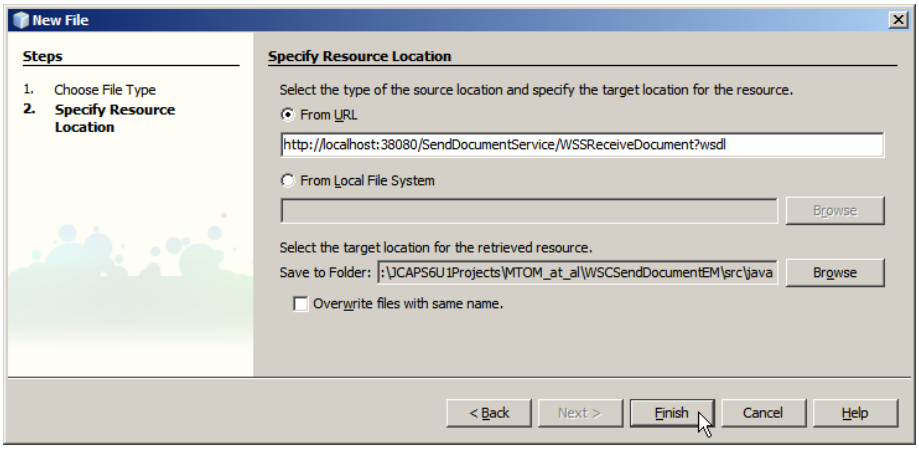

Figure 7-3 Provide WSDL URL and Finish

Rename the WSDL document to WSSReceiveDocumentProxy, as shown in Figure 7-4.

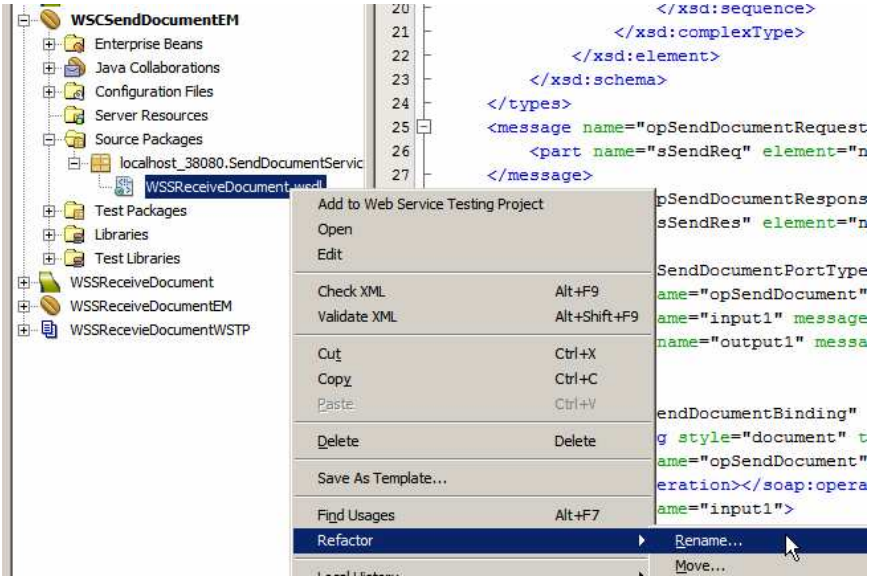

Figure 7-4 Rename the WSDL document

Open the WSDL in WSLD Editor, switch to Source mode and replace all occurrences of SendDocumentPortType with SendDocumentProxyPortType.

Replace all occurrences of SendDocumentBinding with SendDocumentProxyBinding.
Replace all occurrences of SendDocumentPort with SendDocumentProxyPort.
Replace all occurrences of SendDocumentService with SendDocumentProxyService.
Replace the servlet context, /SendDocumentService/WSSReceiveDocument with new servlet context, /SendDocumentServiceProxy/WSSReceiveDocument

Save the modified WSDL, Check and Validate XML.

We now need a Web Service Client reference for the remote web service that the EJB Web Service we are building will invoke. Copy the remote service WSDL URL to the clipboard, as discussed previously. Create a New -> Web Service Client, provide WSDL URL and Finish, as shown in Figure 7-5.

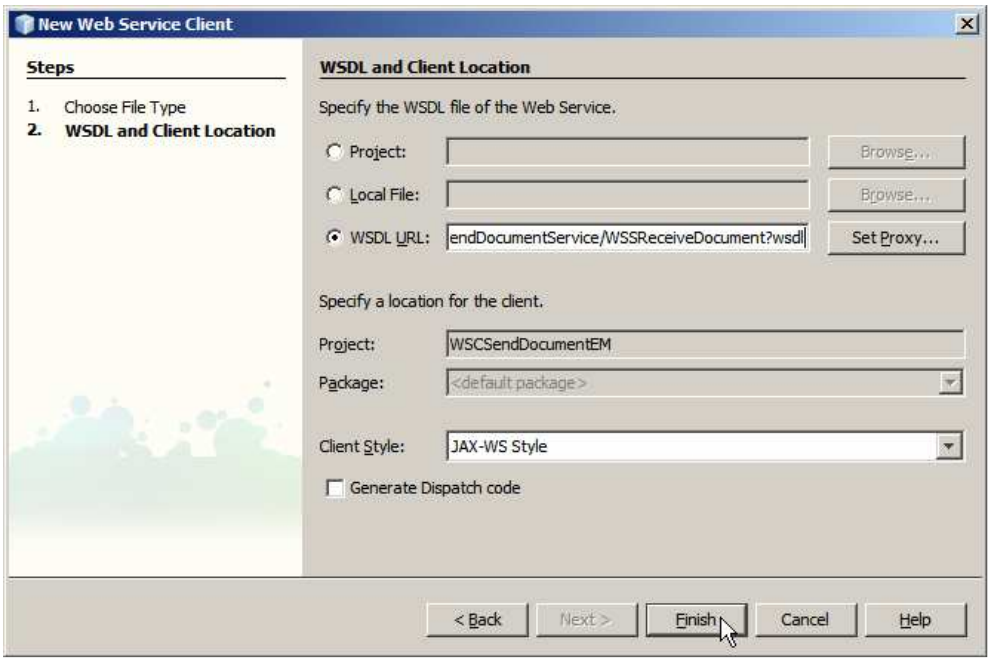

Figure 7-5 Complete creation of a Web Service Client reference

What we will have now will be an EJB Module project with a Web Service Client Reference and a local WSDL. This is shown in Figure 7-6.

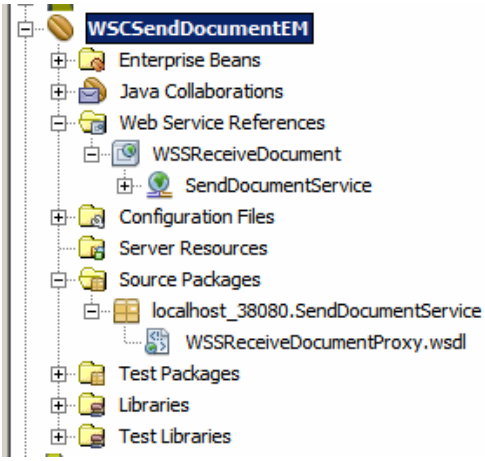

Figure 7-6 EJB Module project with local WSDL and a remote service reference

Let's now create a Web Service implementation from the local WSDL. Right-click on the WSSReceiveDocumentProxy.wsdl, choose Properties, click on the ellipsis next to All Documents property and copy the WSDL location to the clipboard.

Create a Web Service from WSDL. The path is New -> Web Service from WSDL …, or New Project -> Web Services -> Web Service from WSDL.

Name the service WSCReceiveDocumentEM, give package name of pkg. WSCReceiveDocumentEM and paste local WSDL location. This is shown in Figure 7-7.

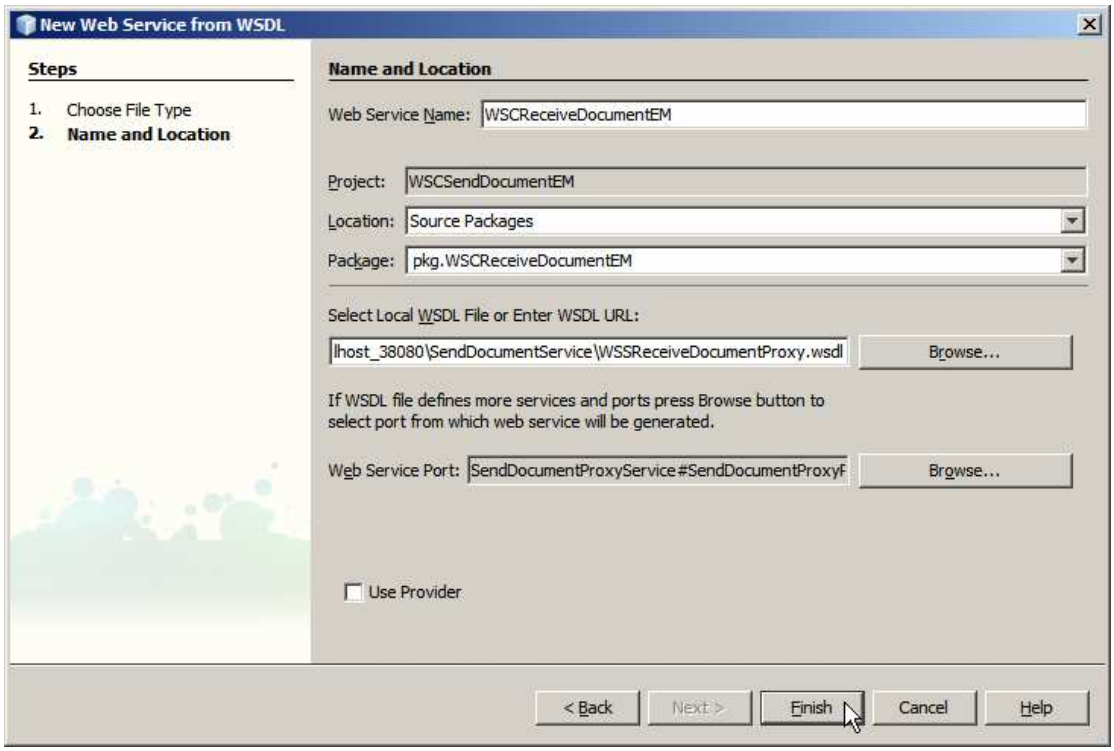

Figure 7-7 Create Web Service from WSDL

Once the web service implementation skeleton is shown in the Design mode make sure to _not_ check the Optimize Transfer of Binary Data (MTOM) checkbox. We don't want to enable MTOM on the Repository-based client to EJB-based client interface. See Figure 7-8.

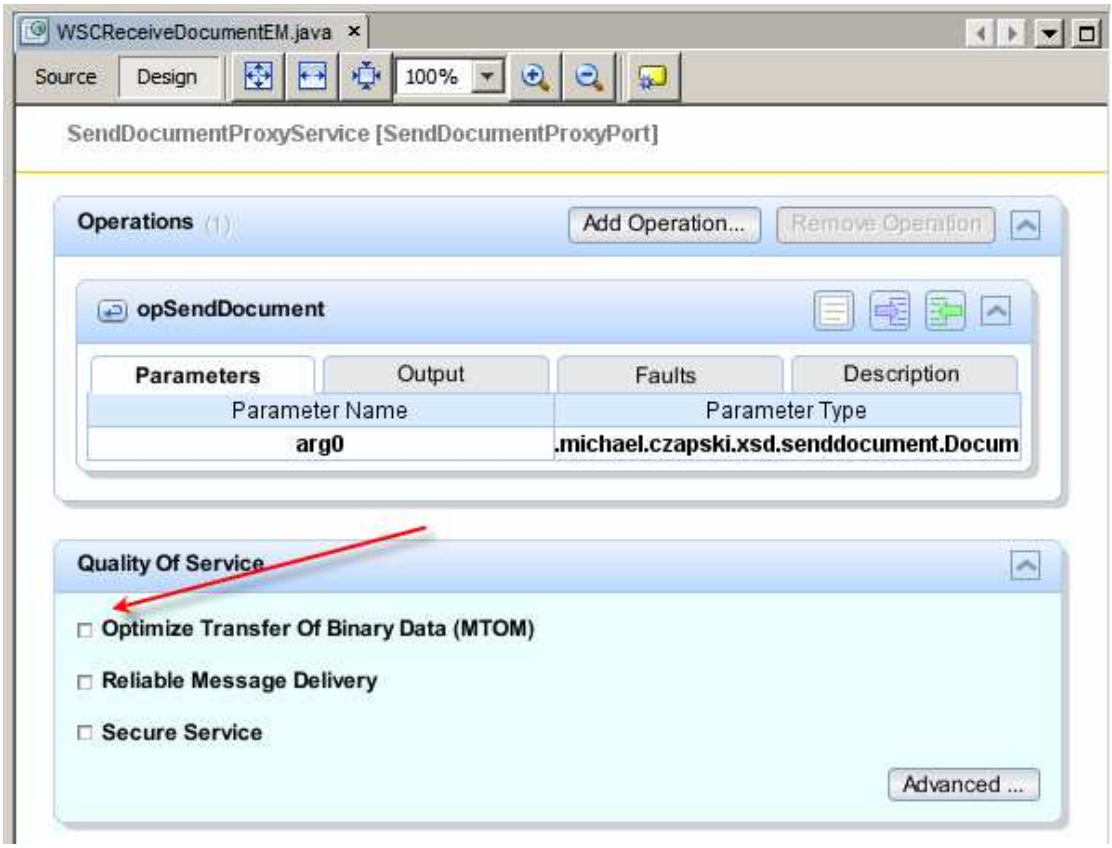

Figure 7-8 Web Service implementation skeleton with _no_ MTOM enabled

Switch to the source code mode, select the two lines which are the body of the opSendDcument method and open an empty line in their place. Expand the Web Service References node tree all the way to the service operation, opSendDocument, then drag the opSendDocument operation onto the source code window as shown in Figure 7-9.

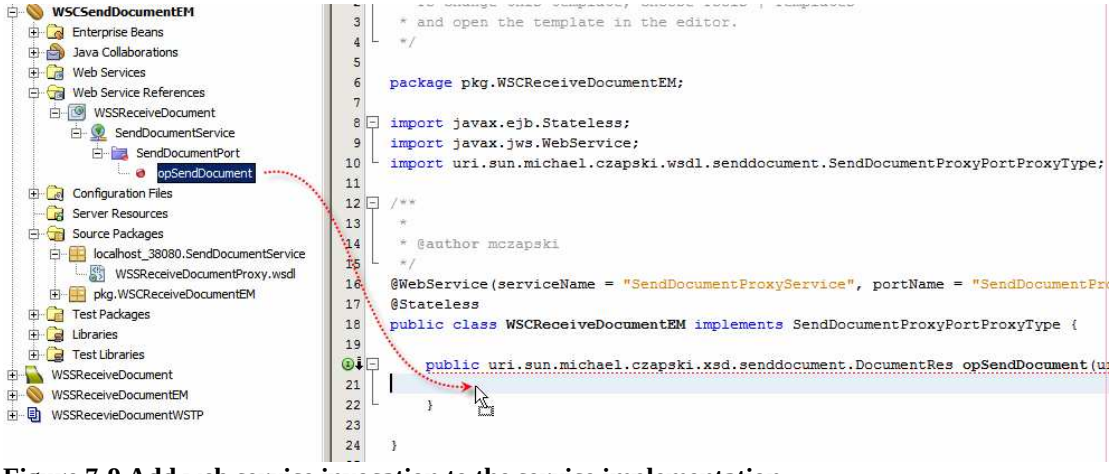

Figure 7-9 Add web service invocation to the service implementation

Modify the body of the opSendDocument method so it reads as shown in Figure 7-10 and Listing 7-1.

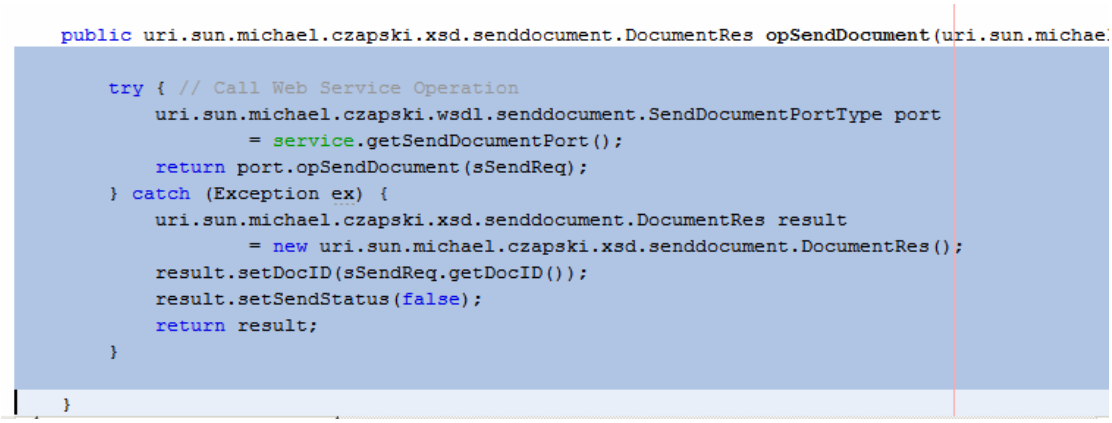

Figure 7-10 Body of the opSendDocument method

Listing 7-1 Body of the opSendDocument method

```
try { // Call Web Service Operation
    uri.sun.michael.czapski.wsdl.senddocument.SendDocumentPortType port
            = service.getSendDocumentPort();
    return port.opSendDocument(sSendReq);
} catch (Exception ex) {
    uri.sun.michael.czapski.xsd.senddocument.DocumentRes result
            = new uri.sun.michael.czapski.xsd.senddocument.DocumentRes();
    result.setDocID(sSendReq.getDocID());
    result.setSendStatus(false);
    return result;
}
```

Build and deploy the web service.

This service does not provide MTOM support, contrary to what the title of this section says. MTOM support will be enabled in this client later, so read on.

8. Exercise EJB-based Web Service Client Wrapper

What we have is a web service implementation. The fact that is invokes another web service is not relevant to this discussion. We need to exercise this web service to ensure it works as expected.

The modern NetBenas IDE, used in Java CAPS 6, provides a number of ways to test web services. The most straight-forward is to install the SoapUI Plugin, which is doable through the Tools->Plugins-accessible Plugin Manager.

Installation of the SoapUI Plugin is discussed in section 15, “

Install soapUI Plugin".

To configure a Web Service Testing Project we will need a URL or a file system location of a WSDL that corresponds to the web service to be tested. The URL we can obtain through the GlassFish Application Server Admin Console, as we have done twice in the previous two sections.

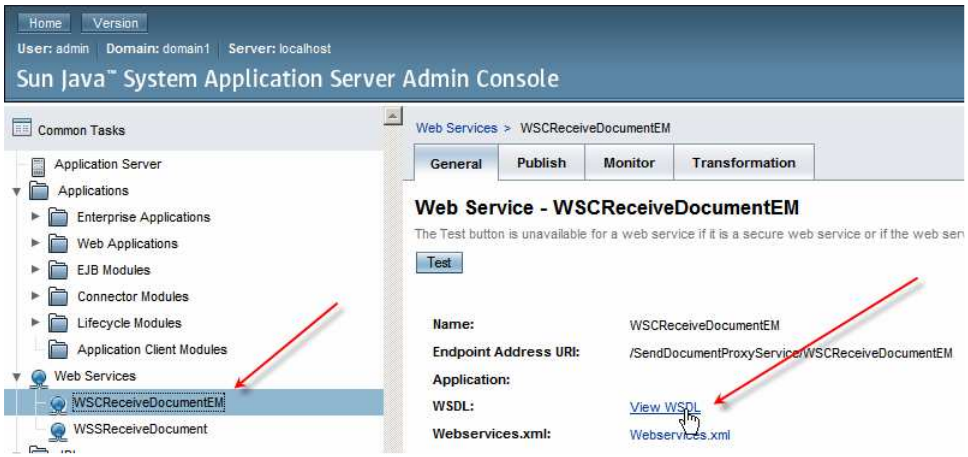

Figure 8-1 Obtain the WSDL URL for the Web Service Client Proxy developed in Section 7

Create a new Web Service testing Project, WSCSendDocumentEMWSTP.

Name the Project and paste the WSDL URL into the Initial WSDL (URL/File).

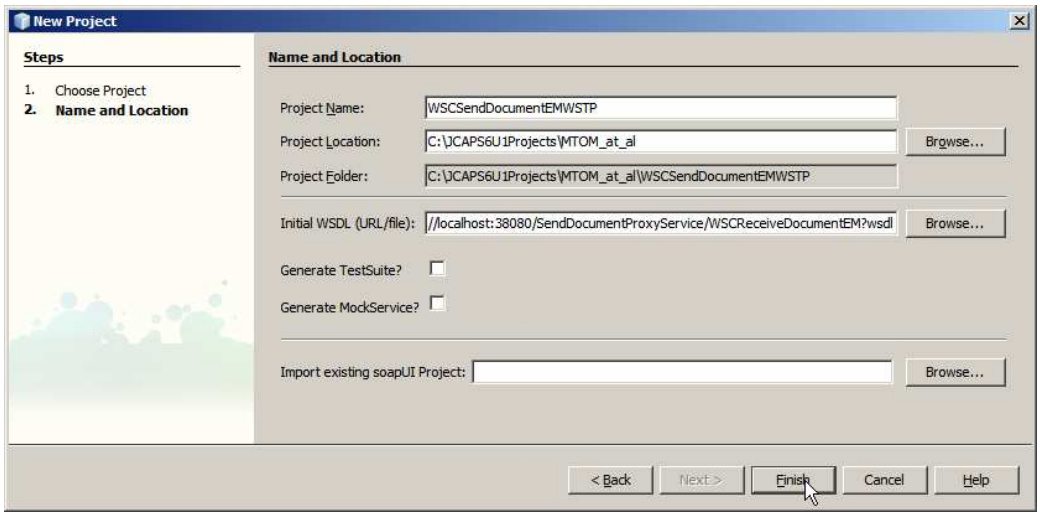

Figure 8-2 Paste the URL into the box.

Once the WSDL is parsed and imported, expand the project structure and open "Request 1" for editing.

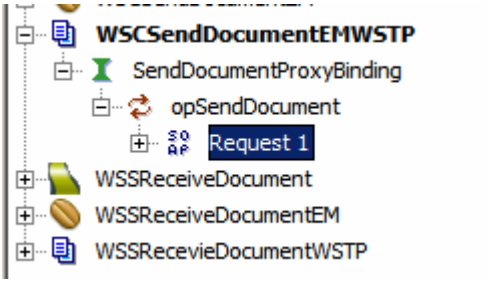

Figure 8-3 Open Request 1 for editing

Complete the request by providing data for the fields. Paste the following string, not including quotes, into the optional DocBody "ZHVtbXk=". Submit the request.

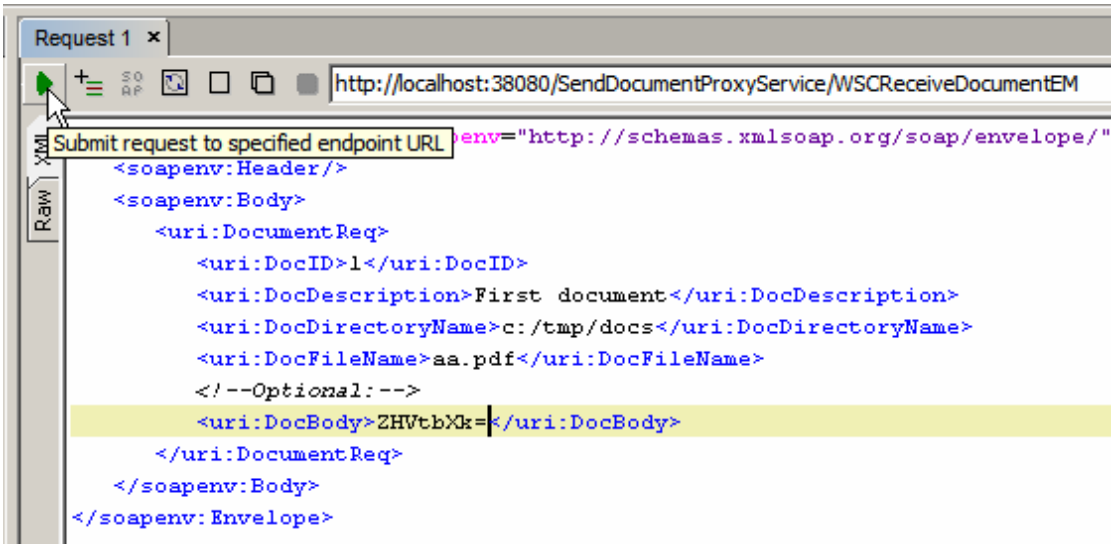

Figure 8-4 Configure SOAP Request

The response returned by the service should look like this:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:enc="ht
   <env:Body>
      <ns0:DocumentRes xmlns:tns="uri:Sun:Michael:Czapski:XSD:SendDocument">
         <tns:DocID>1</tns:DocID>
         <tns:SendStatus>true</tns:SendStatus>
      </ns0:DocumentRes>
   </env:Body>
</env:Envelope>
```

Figure 8-5 SOAP Response

Let's now start the TCP Mon as a proxy listening on port 8888, see section 14, "

Obtain and use the Apache TCP Mon", and configure NetBeans to use that as the Web Proxy. Tools->Options->General: Manual Proxy Settings.

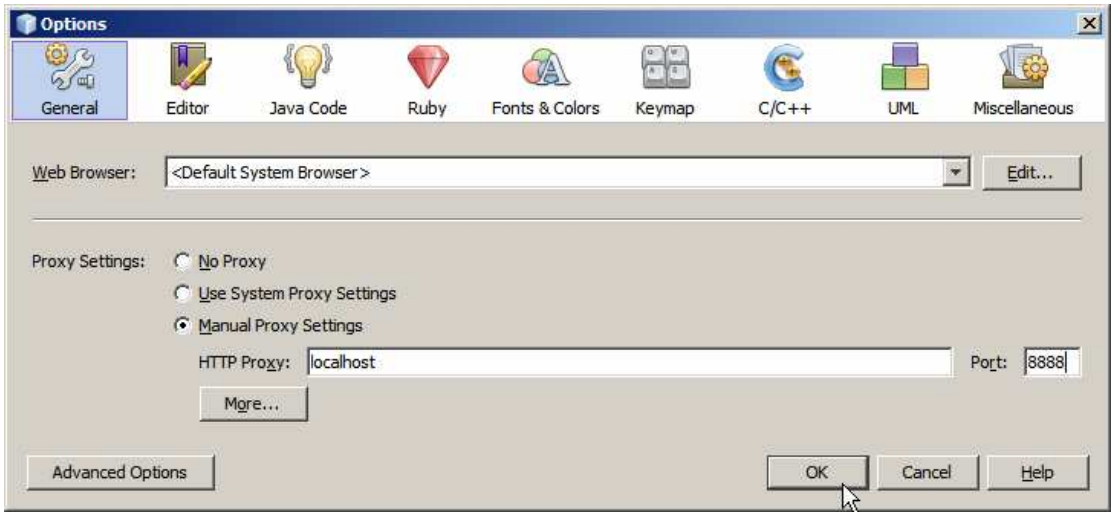

Figure 8-6 Configure Web Proxy

Submit the request again and view the TCP Mon display, noticing the SOA Request and the SOAP Response.

```
POST http://localhost:38080/SendDocumentProxyService/WSCReceiveDocumentEM HTTP/1.1
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
User-Agent: Jakarta Commons-HttpClient/3.1
Host: localhost:38080
Proxy-Connection: Keep-Alive
Content-Length: 543

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:uri="uri:Sun:Michael:Czapski:XSD:SendDocument">
   <soapenv:Header/>
   <soapenv:Body>
      <uri:DocumentReq>
         <uri:DocID>1</uri:DocID>
         <uri:DocDescription>First document</uri:DocDescription>
         <uri:DocDirectoryName>c:/tmp/docs</uri:DocDirectoryName>
         <uri:DocFileName>aa.pdf</uri:DocFileName>
         <!--Optional:-->
         <uri:DocBody>ZHVtbXk=</uri:DocBody>
      </uri:DocumentReq>
   </soapenv:Body>
</soapenv:Envelope>
```

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: multipart/related;start="<rootpart*d99fd774-26e6-4d7b-8fac-77261cd4084d@example.jaxws.sun.com>";type="application/xo
Transfer-Encoding: chunked
Date: Sat, 14 Feb 2009 04:55:45 GMT

1d3
--uuid:d99fd774-26e6-4d7b-8fac-77261cd4084d
Content-Id: <rootpart*d99fd774-26e6-4d7b-8fac-77261cd4084d@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><DocumentRes xmlns="uri:Sun:Michael:
2f

--uuid:d99fd774-26e6-4d7b-8fac-77261cd4084d--
```

Figure 8-7 SOAP Request (top) and SOAP Response (bottom)

MTOM optimisation is not evident on the request. This is because we did not configure MTOM support on the client side in the EJB Web Service client. We will do that later.

9. Exercise the Repository Web Service Client - 2

Let's point the Repository-based Web Service Client at the EJB-based Web Service Wrapper Client, built and exercised in the previous two sections.

The schematic in Figure 9-1 illustrates the interactions.

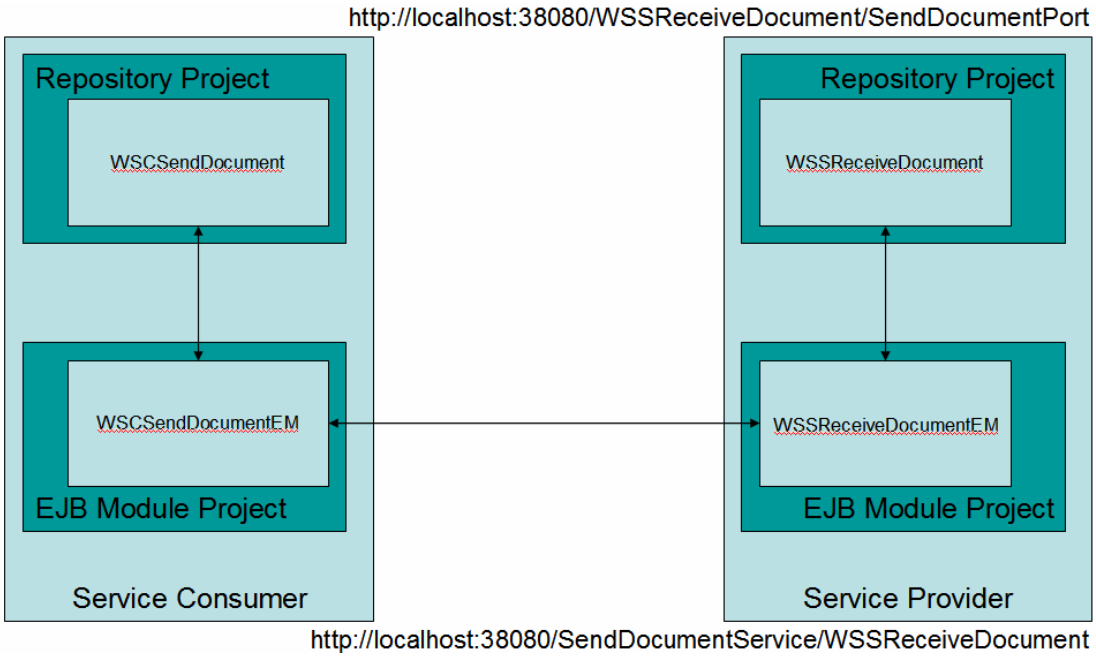

Figure 9-1 Final interaction

Let's copy the servlet context part of the WSDL URL of the Web Service Client Proxy to the clipboard. Figure 9-2 illustrates the key point.

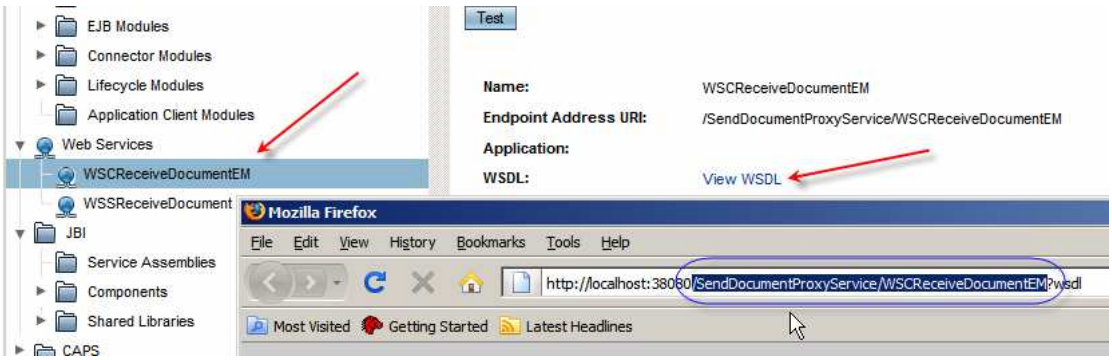

Figure 9-2 Copy the servlet context of the web service proxy

Let's open the properties of the WSCSendDocument external system container in the CAPS Environment, WSEnv, and replace the servlet context property value with the value copied to the clipboard. Figure 9-3 illustrates this.

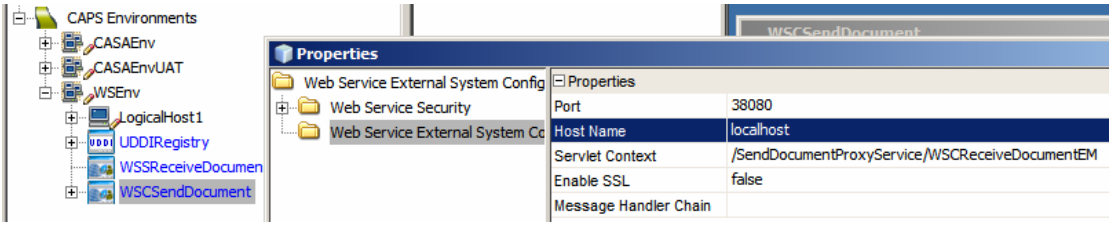

Figure 9-3 Change the servlet context and the port

Let's also undo the setting to the HTTP Proxy properties. We will not be using the proxy this time. Figure 9-4 illustrates the setting as they ought to be now.

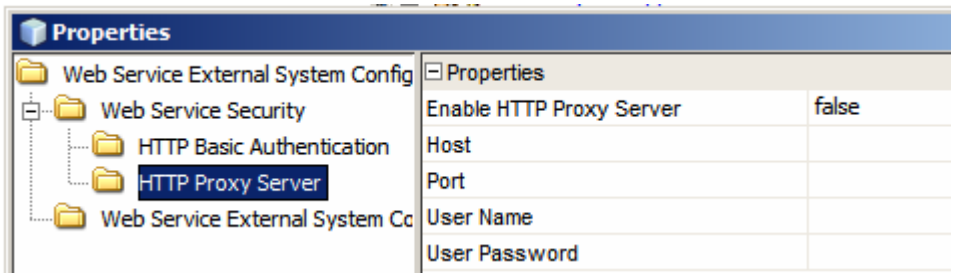

Figure 9-4 Undo proxy settings

Let's build and deploy the project.

Let's now submit a JMS message, perhaps with the value 6, and observe the response in the output Queue. Figure 9-5 illustrates this.

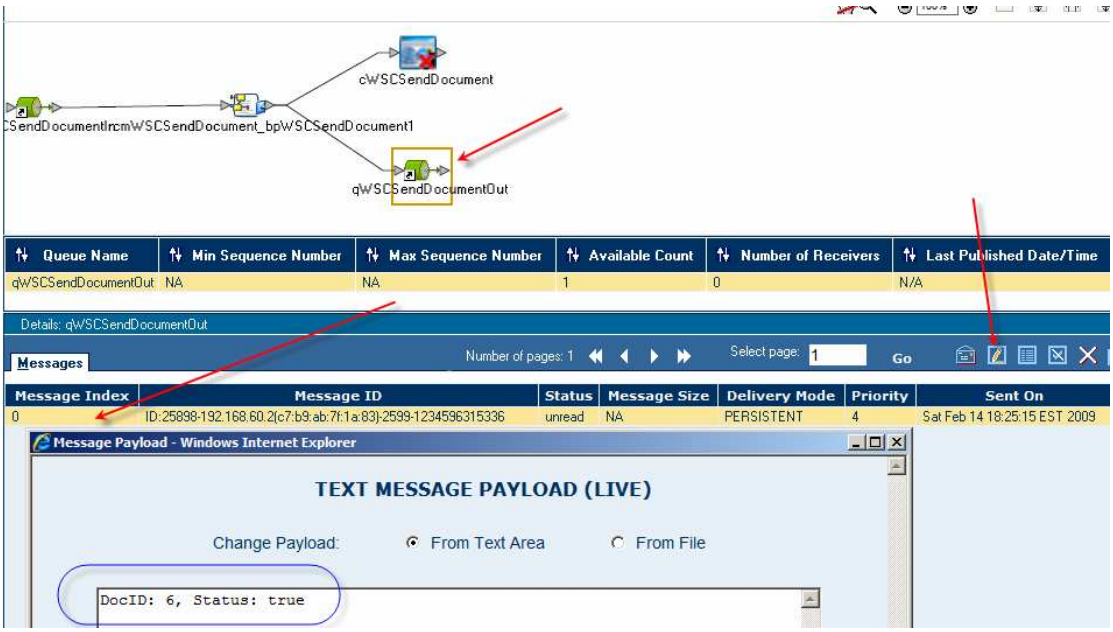

Figure 9-5 Web Service response, passed through the solution

We cannot observe on-the-wire messages until we rig up the client proxy so that it submits requests to the TCP Mon listener instead of the real service provider.

Let's start the TCP Mon such that it listens on port 38081 and redirects to the port on which the real web service listens. For me this is 38080.

The commands might be:

```
C:> cd C:\tools\tcpmon-1.0-bin\build
C:> tcpmon.bat 38081 localhost 38080
```

Let's copy the WSDL URL of the remote service, WSSReceiveDocument, to the clipboard using the GlassFish Applicatin Server Admin Console, much as we have done previously. See Figure 9-6.

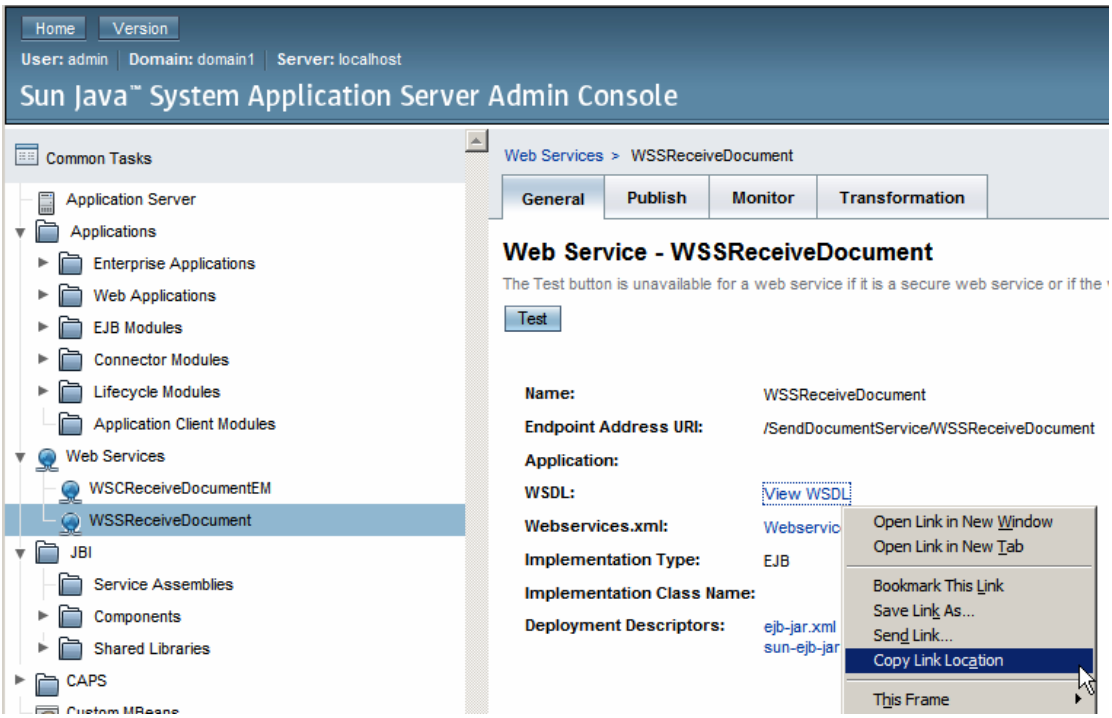

Figure 9-6 Copy Web Service WSDL URL to the clipboard.

Right-click on the WSSReceiveDocument service under the Web Services References node of the WSCSendDocumentEM project, and choose Refresh Client as shown in Figure 9-7.

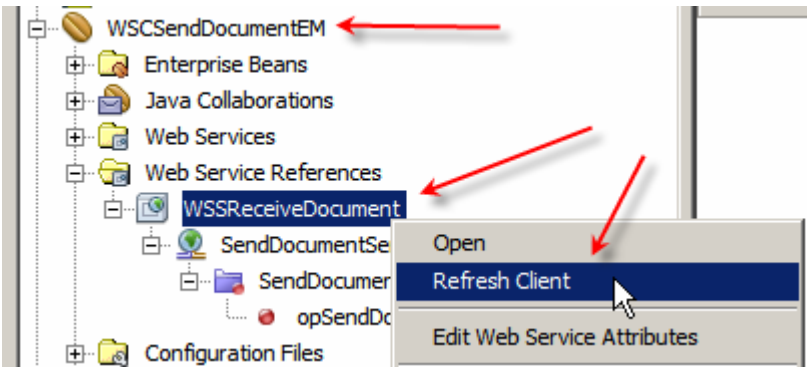

Figure 9-7 Trigger Refresh Client funcitonality

Check the "Also replace local wsdl file with the original …" checkbox and make sure to change the port number from the original to that of the TCP Mon listener – for me a change from 38080 to 38081. Figure 9-8 illustrates this.

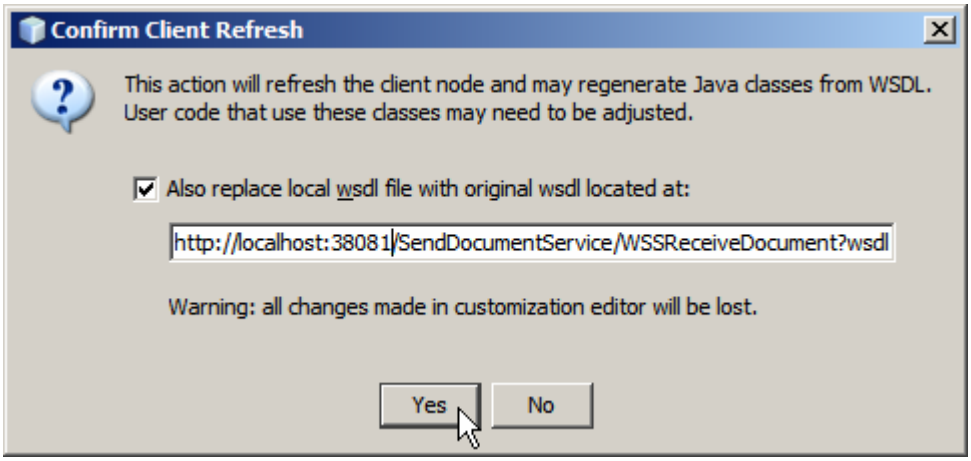

Figure 9-8 Refresh the WSDL making sure it is retrieved through the TCP Mon proxy

If you switch to the TCP Mon window you will see the WSDL retrieved and, very importantly, the Endpoint URL modified to use the TCP Mon listener port rather then the original service port. Figure 9-9 shows this.

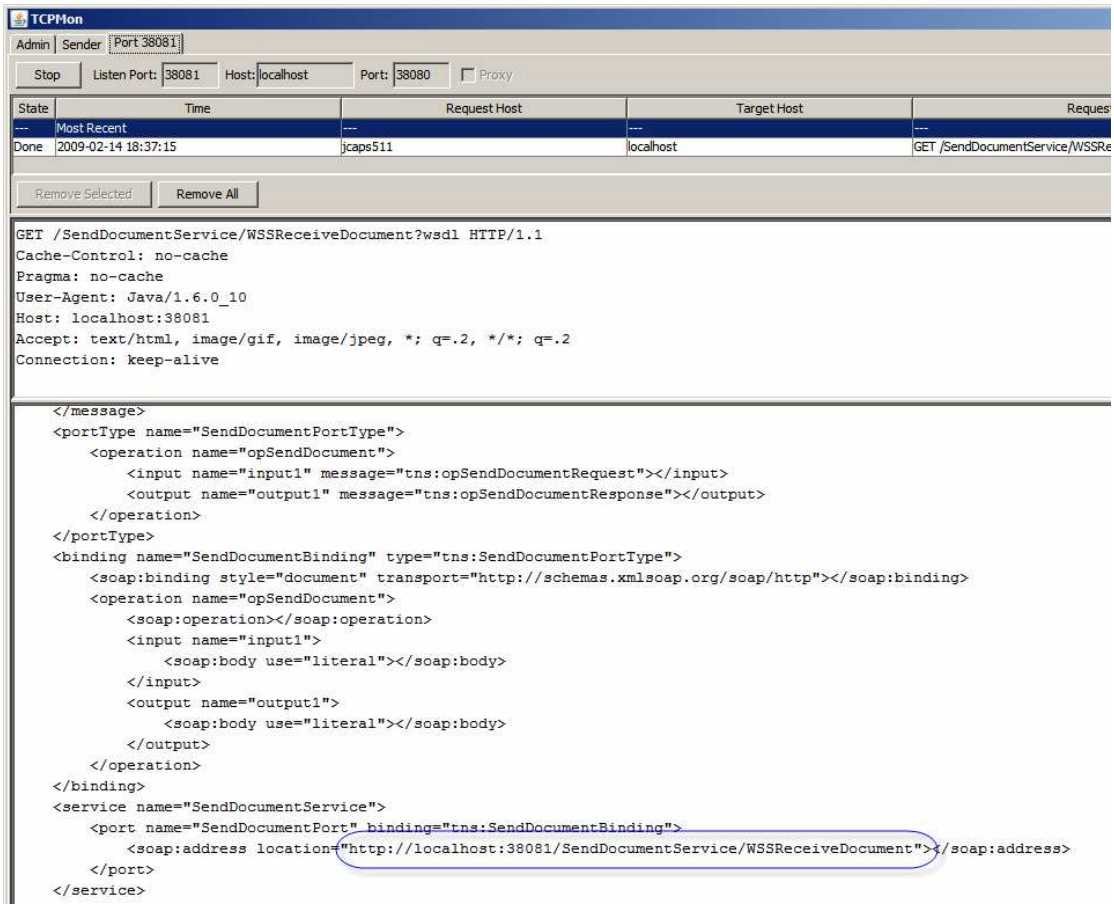

Figure 9-9 Service WSDL retrieved through the TCP Mon proxy

Open the service implementation in Source mode and change the port number to that of the TCP Mon listener, as shown in Figure 9-10.

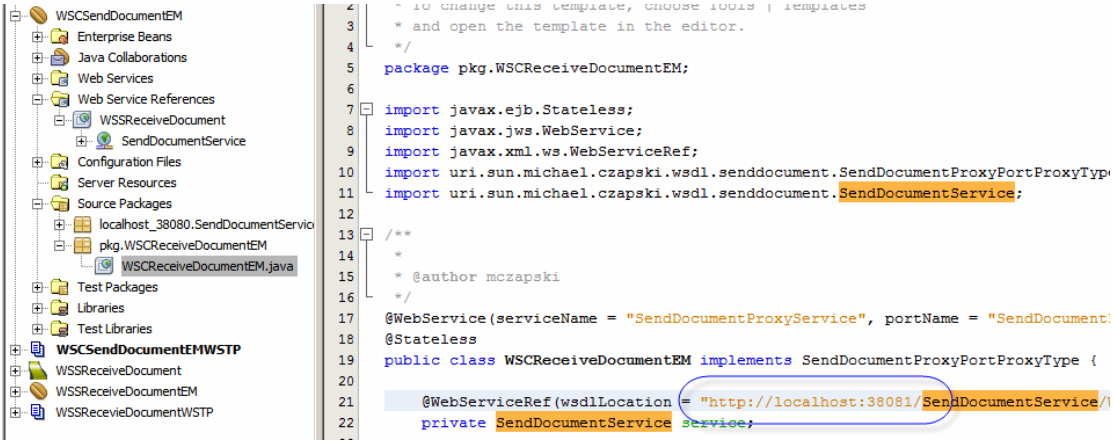

Figure 9-10 Change WSDL port number to use the TCP Mon Listener

Clean and Build, the project, deploy the project, WSCSendDocumentEM, then submit a message to the JMS Queue qWSCSendDocumentIn, perhaps with the value of 7.

Observe the response in the JMS Queue qWSCSendDocumentOut.

Switch to the TCP Mon and observe the Request and the Response. Notice that no MTOM optimisation has taken place.

10. Add Client-side MTOM support to EJB Client

To have the Client perform MTOM optimisation source code change are required.

Open the WSCReceiveDocumentEM.java in source code mode. Figure 10-1 points out where the Java source lives.

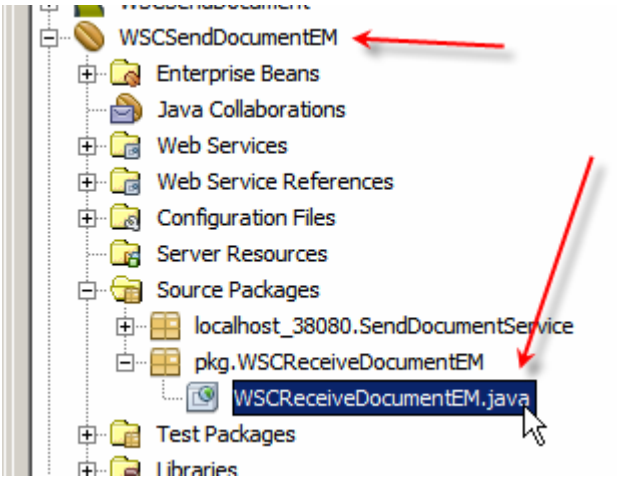

Figure 10-1 Locating the Java source

Add the following import statement after the last import statement:

```
import javax.xml.ws.soap.MTOMFeature;
```

Insert the following code between the parentheses of the `service.getSendDocumentPort()` method invocation:

```
new MTOMFeature()
```

The code fragment should read: service.getSendDocumentPort(new MTOMFeature())

Figure 10-2 shows the source code after the changes.

```
import javax.jws.WebService;
import javax.xml.ws.WebServiceRef;
import uri.sun.michael.czapski.wsdl.senddocument.SendDocumentProxyPortProxyType;
import uri.sun.michael.czapski.wsdl.senddocument.SendDocumentService;
import javax.xml.ws.soap.MTOMFeature;

/**
 *
 * @author mczapski
 */
@WebService(serviceName = "SendDocumentProxyService", portName = "SendDocumentPr
@Stateless
public class WSCReceiveDocumentEM implements SendDocumentProxyPortProxyType {

    @WebServiceRef(wsdlLocation = "http://localhost:38081/SendDocumentService/WS
    private SendDocumentService service;

    public uri.sun.michael.czapski.xsd.senddocument.DocumentRes opSendDocument(u

try { // Call Web Service Operation
    uri.sun.michael.czapski.wsdl.senddocument.SendDocumentPortType port
            = service.getSendDocumentPort(new MTOMFeature());
    return port.opSendDocument(sSendReq);
} catch (Exception ex) {
```

Figure 10-2 Code modifications required to enforce MTOM support

Build and deploy the project.

Submit a JMS message, perhaps containing the value of 12, and observe the Request and the Response on-the-wire in the TCP Mon window. The Request and the Response in Figure 10-3 have been reformatted manually to improve readability. Note that the DocBody, which started as a Base64-encoded string in the Repository-based Web Service Client has been extracted, decoded, and appended as a MIME part. The DocBody content now contains the references to the MIME part carrying a binary content of the DocBody the service proxy received. The response indicates that the message was received successfully. The JMS message in the Queue qWSCSendDocumentOut also shows success.

```
--uuid:9754eb99-f0a7-47ee-a6ea-e31e708dcb08
Content-Id: <rootpart*9754eb99-f0a7-47ee-a6ea-e31e708dcb08@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<DocumentReq xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument">
<DocID>12</DocID>
<DocDescription>Document ID: 12</DocDescription>
<DocDirectoryName>c:\tmp\docs</DocDirectoryName>
<DocFileName>bb.pdf</DocFileName>
<DocBody>
       <Include xmlns="http://www.w3.org/2004/08/xop/include"
                href="cid:aefcfea6-96d1-4e4a-8ced-d5385f7680db@example.jaxws.sun.com"/>
</DocBody>
</DocumentReq>
</S:Body>
</S:Envelope>
--uuid:9754eb99-f0a7-47ee-a6ea-e31e708dcb08
Content-Id: <aefcfea6-96d1-4e4a-8ced-d5385f7680db@example.jaxws.sun.com>
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

This is doc body
--uuid:9754eb99-f0a7-47ee-a6ea-e31e708dcb08--
```

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Sat, 14 Feb 2009 08:57:46 GMT

ee
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<DocumentRes xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument">
<DocID>12</DocID><SendStatus>true</SendStatus></DocumentRes></S:Body></S:Envelope>
```

Figure 10-3 MTOM-optimised request on-the-wire

11. Watching the Wire without TCP Mon

To watch what is happening on the wire without using the TCP Mon one can add:

```
-Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true
```

to the GlassFish Application Server JVM Options and restart the Application Server.

Submitting a JMS Message, perhaps with the value of 11, will show the following in the GlassFish server.log:

Listing 11-1 Request from WSCSendDocument to WSCSendDocumentEM

```
[#|2009-02-14T20:24:13.071+1100|INFO|sun-
appserver9.1|javax.enterprise.system.stream.out|_ThreadID=35;_ThreadName=httpSSLWorker
Thread-38080-1;|
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns0="uri:Sun:Michael:Czapski:XSD:SendDocument"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <ns0:DocumentReq xmlns:tns="uri:Sun:Michael:Czapski:XSD:SendDocument">
      <tns:DocID>11</tns:DocID>
      <tns:DocDescription>Document ID: 11</tns:DocDescription>
      <tns:DocDirectoryName>c:\tmp\docs</tns:DocDirectoryName>
      <tns:DocFileName>bb.pdf</tns:DocFileName>
      <tns:DocBody>VGhpcyBpcyBkb2MgYm9keQ==</tns:DocBody>
    </ns0:DocumentReq>
  </env:Body>
</env:Envelope>
|#]
```

Listing 11-2 Request from WSCSendDocumentEM to WSSReceiveDocumentEM

```
[#|2009-02-14T20:24:13.774+1100|INFO|sun-
appserver9.1|javax.enterprise.system.stream.out|_ThreadID=36;_ThreadName=httpSSLWorker
Thread-38080-0;|
--uuid:1df04f68-f019-4a31-872d-8a3546742ea6
Content-Id: <rootpart*1df04f68-f019-4a31-872d-8a3546742ea6@example.jaxws.sun.com>
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body><DocumentReq xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument"><DocID>11</DocID
><DocDescription>Document ID: 11</DocDescription><DocDirectoryName>c:\tmp\docs</DocDir
ectoryName><DocFileName>bb.pdf</DocFileName><DocBody><Include
xmlns="http://www.w3.org/2004/08/xop/include"
href="cid:3bf57164-ac9f-45ca-87ab-047c69fa889e@example.jaxws.sun.com"/></DocBody></Doc
umentReq></S:Body></S:Envelope>
--uuid:1df04f68-f019-4a31-872d-8a3546742ea6
Content-Id: <3bf57164-ac9f-45ca-87ab-047c69fa889e@example.jaxws.sun.com>
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

This is doc body
--uuid:1df04f68-f019-4a31-872d-8a3546742ea6--|#]
```

Listing 11-3 Request from WSSReceiveDocuemtnEM to WSSReceiveDocument

```
[#|2009-02-14T20:24:13.805+1100|INFO|sun-
appserver9.1|javax.enterprise.system.stream.out|_ThreadID=36;_ThreadName=httpSSLWorker
Thread-38080-0;|
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <DocumentReq xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument">
      <DocID>11</DocID>
      <DocDescription>Document ID: 11</DocDescription>
      <DocDirectoryName>c:\tmp\docs</DocDirectoryName>
```

```
      <DocFileName>bb.pdf</DocFileName>
      <DocBody>VGhpcyBpcyBkb2MgYm9keQ==</DocBody>
    </DocumentReq>
  </S:Body>
</S:Envelope>
|#]
```

Listing 11-4 Response from WSSReceiveDocument to WSSReceiveDocumentEM

```
[#|2009-02-14T20:24:17.914+1100|INFO|sun-
appserver9.1|javax.enterprise.system.stream.out|_ThreadID=36;_ThreadName=httpSSLWorker
Thread-38080-0;|
====[com.sun.xml.ws.assembler.server.transport:response]====|#]

[#|2009-02-14T20:24:17.914+1100|INFO|sun-appserver9.1|javax.enterprise.system.stream.o
ut|_ThreadID=36;_ThreadName=httpSSLWorkerThread-38080-0;|
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <DocumentRes xmlns="uri:Sun:Michael:Czapski:XSD:SendDocument">
      <DocID>11</DocID>
      <SendStatus>true</SendStatus>
    </DocumentRes>
  </S:Body>
</S:Envelope>
|#]
```

As can be clearly seen the base64binary request fields are extracted, attached as unencoded binary MIME parts and referenced back in the original request. This is how MTOM Operates.

12. Summary

It is fairly easy to implement MTOM support for Java CAPS 6 repository-based web services both at the client and at the service side by creating EJB-based Web Service Wrappers. It is still stupid to design solutions in which large body of data is transferred using SOAP Requests and Responses. MTOM optimisation only helps with the wire transfer in that the binary data is transferred as binary data, rather then as base64-encoded data as would have been the case without MTOM. The fact remains that internal conversion from binary to Base64 and back again happens at both the client and the service side. That has a cost associated with it.

The method discussed here is OK is we have existing Repository-based Web Service Client or Provider and need to add MTOM optimisation without changing the Repository-based web service implementations. It may be more efficient to have the Repository-based components use JSM to communicate with the EJB Wrappers. This can be achieved by creating EJB-based JCA MDBs that send/receive JMS messages on the one side and send/receive SOAP Responses/Request on the other. This may or may not be a subject of another Note

13. Create Java CAPS Environment

Create a Java CAPS Environment, WSEnv, as illustrated in Figures Figure 13-1 through Figure 13-3.

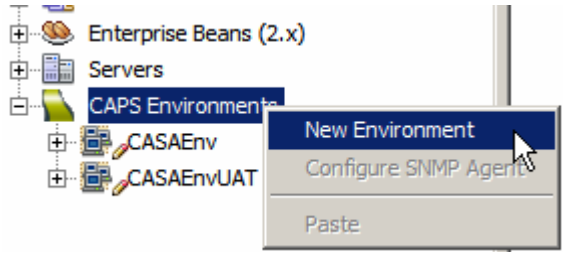

Figure 13-1 Create Java CAPS Environment

Add a Logical Host. Add a Sun Java System Application Server and a Sun Java System Message Queue. Set the properties for both to provide authentication credentials, host names and port numbers that reflect your environment.

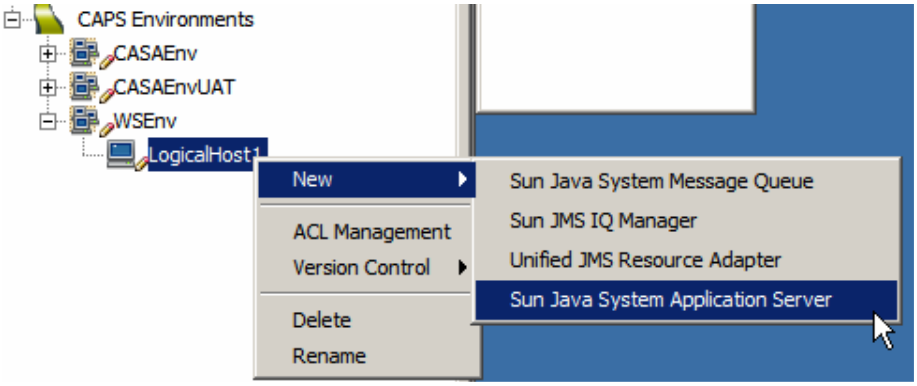

Figure 13-2 Add an Application Server to the Environment's Logical Host.

Add a new UDDI External System container and modify its properties to reflect your environment.

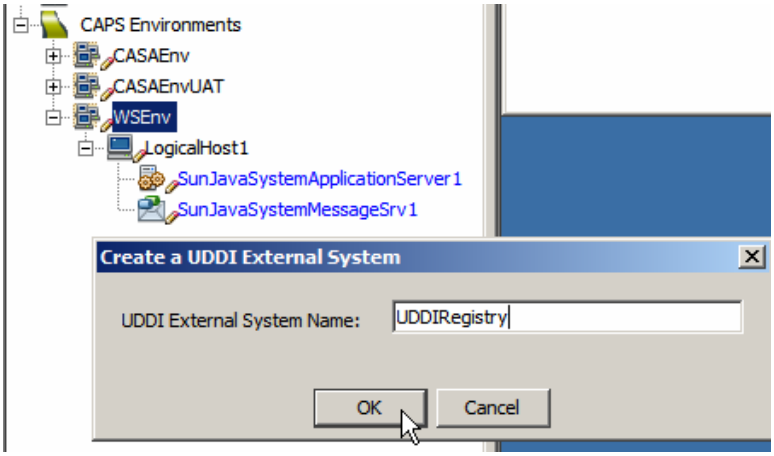

Figure 13-3 Add new UDDI External System

We will add and configure Web Services Client and Server External System containers as we go along in solution development.

14. Obtain and use the Apache TCP Mon

One of the issues with SOAP decoration, which is what MTOM and others do, is that the on-the-wire message looks different from what the sending and the receiving applications see. It is perhaps harder then necessary to make the application server and the client log what they are sending and receiving and even then there is a good chance that what is logged differs from what is sent / received. To see what is really exchanged a wire snooper of some sort will come handy.

Apache TCP Mon, see http://ws.apache.org/commons/tcpmon/index.html, can be used as a convenient proxy to view the on-the-wire messages exchanged between web services invokers and providers. Download the TCP Mon from the site. Tutorial at http://ws.apache.org/commons/tcpmon/tcpmontutorial.html has a nice explanation of the usage modes.

To start the TCP Mon from the command line in a direct intermediary mode with specific host and port configuration one could say (on Windows):

```
C:> cd C:\tools\tcpmon-1.0-bin\build
C:> tcpmon.bat 38081 localhost 38080
```

This will start the TCP Mon with the listening port 38081, relaying messages to port 38080 on localhost.

```
C:> cd C:\tools\tcpmon-1.0-bin\build
C:> tcpmon.bat 8888
```

This will start the TCP Mon as a proxy listening on port 8888. One needs to configure one's client to use the proxy.

15. Install soapUI Plugin

To test EJB-based web services with complex messages SoapUI plugin needs be installed.

Tools->Plugins->Available Plugins

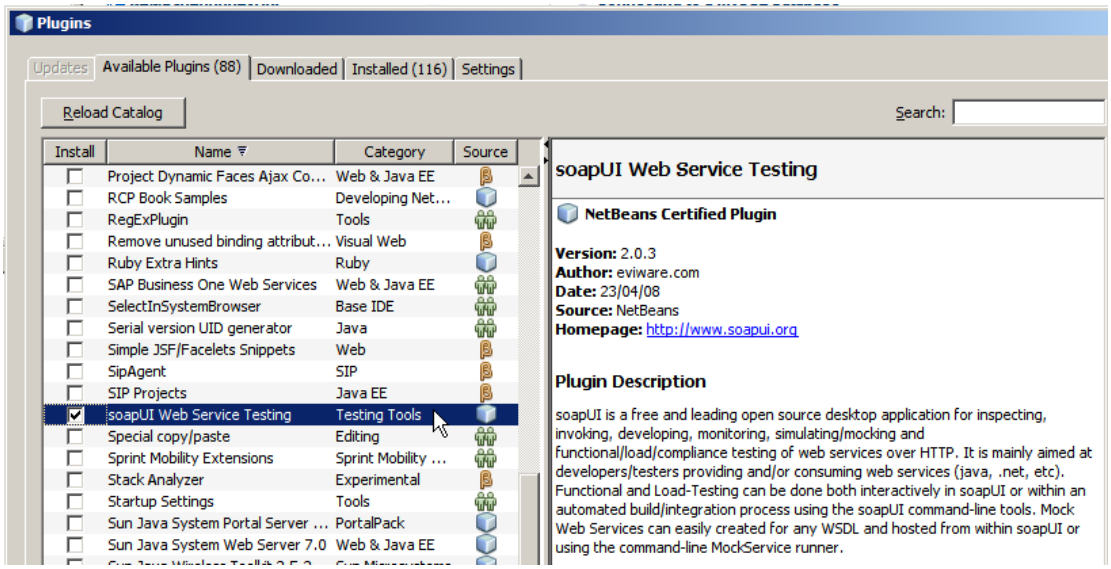

Figure 15-1 Locate soapUI in the list of AvailablePlugins

Click the checkbox next to plugin name and click Install. Follow the prompts.

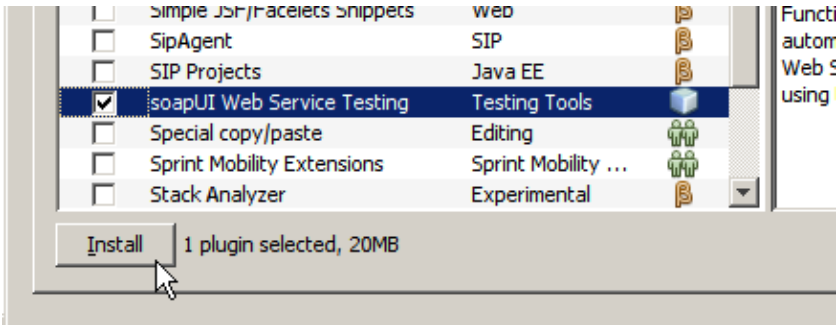

Figure 15-2 Choosing to install the plugin

You will need to have access to the Internet as the tooling will try to download the plugin.